

嵌入式技术与应用丛书

ARM处理器裸机开发实战

——机制而非策略

王小强 主
李英花 方 鹏 副主
粟思科 审 校

◎ 内容全面。涉及ARM处理器的基本概念，并附有相应的实例解析
◎ 实用性强。以基础理论的介绍为主，同时给出具体的案例介绍和分析
◎ 概念清晰，内容翔实，体系合理



電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

華信 欢迎登录 免费 获取优质教学资源
http://www.hxedu.com.cn

ARM处理器裸机开发实战

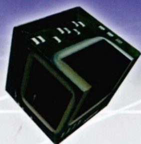
——机制而非策略

写作特点

★ 理论和实践相结合，夯实理论基础，强化实践环节
★ 模块化设计与系统设计相结合
★ 立足于ARM处理器，并给出了设计思想与方案
★ 用朴实的语言描述看似深奥的理论

适用对象

★ 高等院校电子、通信、自动控制等专业学生
★ 从单片机开发向ARM嵌入式开发转型的工程师
★ 从事ARM嵌入式开发的相关技术人员



策划编辑：曲　昕

责任编辑：徐云鹏
封面设计：世纪创典

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。





ISBN 978-7-121-15303-7
9 787121 153037 >
定价：56.00 元
（含视频学习光盘1张）

嵌入式技术与应用丛书

ARM处理器裸机开发实战
——机制而非策略

王小强　主　编
李英花　方　鹏　副主编
粟思科　审　校

電子工業出版社
Publishing House of Electronics Industry
北京・BEIJING

内容简介

本书面向立志于进行ARM嵌入式开发的初学者以及从单片机向ARM处理器转型的工程师，按照理论实践相结合的思想，介绍了ARM嵌入式开发过程中的基础理论，并给出了具体的实例。全书共分为4篇，包括ARM汇编语言、ARM C语言、ARM处理器各功能模块开发等内容。

本书针对ARM处理器裸机开发过程中的重点、难点问题，既有基础知识的讲述，又有相关配套实验，使读者能容易、快速、全面地掌握ARM处理器裸机开发。

本书循序渐进、内容完整、实用性强，以教材方式组织内容，可作为高等院校电子、通信、自动控制等专业的学习用书，也可供广大嵌入式工程师作为参考。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

ARM处理器裸机开发实战：机制而非策略 / 王小强主编. —北京：电子工业出版社，2012.1
（嵌入式技术与应用丛书）
ISBN 978-7-121-15303-7

Ⅰ. ①A… Ⅱ. ①王… Ⅲ. ①微处理器，ARM－系统设计 Ⅳ. ①TP332

中国版本图书馆CIP数据核字（2011）第243376号

策划编辑：曲 昕
责任编辑：徐云鹏　　特约编辑：张燕虹
印　　刷：
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路173信箱　邮编　100036
开　　本：787×1 092　1/16　印张：24　字数：614千字
印　　次：2012年1月第1次印刷
定　　价：56.00元（含视频学习光盘1张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

序



——机制而非策略

在 UNIX/Linux 的接口设计方面有个很好的理念“提供机制而非策略”。其实，大部分的软件编程问题都可以划分成两个部分：“需要提供哪些功能”（机制）和“怎样实现这些功能”（策略）。如果程序中分别由不同的模块负责机制和策略的实现，那么软件开发就更容易，也更有利于应用到不同的应用环境中。

机制与策略问题更反映了这样一个理念：简单就是美，力图把复杂的问题简单化，而不是把简单问题复杂化。这一理念对于一个刚刚开始学习 ARM 开发的人员来说尤其重要，ARM 处理器有其自身的复杂性，初学者既要学习 ARM 开发工具，又要学习编译链接的基础知识。此外，对于学习基于 ARM 的嵌入式系统开发而言，还需要懂得 ARM 处理器各部分硬件的基础知识和相应的操作原理，需要结合具体的实验来掌握硬件的控制方法，因为理论毕竟是理论，其中省略了很多细节的东西，而这些细节的问题虽然表面上看起来没有太多的理论价值，但正是这些细节的问题始终困扰着初学者。例如，《ARM920T Technical Reference Manual》、《ARM Architecture Reference Manual》、《ARM Target Development System—User's Guid》、《ARM Software Development Toolkit—Reference Guid》等文档确实堪称是 ARM 开发的宝典（可以在 ARM 官网上下载），但即使读者通读这些文档（如果能读懂的话），可能最后连什么是软中断，如何在裸机上面实现软中断等问题都解决不了。

与其将 ARM 处理器当做一款处理器来学习，不如将其当做一款功能强大的单片机来掌握，并不是一开始就学习复杂的地址映射关系，也不是一开始就学习 ARM 可执行文件的格式，而是尝试着简简单单地控制一个寄存器，控制一个 I/O 引脚，点亮一个 LED，学习角度变化以后，会使问题简单化，当读者一旦掌握了处理器外围器件的控制以后，再学习 Uboot 移植、Linux 内核移植、文件系统移植、驱动开发的时候，就会发现自己已经慢慢地步入了 ARM 开发的行列（当然只是刚刚起步，真正的技术是在一个个具体项目中锻炼出来的）。

本书采用的学习路线是：在 Windows 环境下用 ADS 1.2 进行 ARM 裸机程序的开发，掌握到一定程度后，然后转到 Linux 环境下进行 Uboot 移植、操作系统移植、文件系统移植、GUI 移植和驱动程序的开发等，本书只涉及了 ADS 1.2 环境下 ARM 裸机程序的开发。在裸机开发过程中，读者可能会遇到很多问题，如什么是启动代码，什么是 SDRAM，什么是 NAND FLASH，什么是 NOR FLASH、SDRAM 控制器、NAND FLASH 控制器和 NOR FLASH 控制器各有什么作用，从 NAND FLASH 启动和 NOR FLASH 启动有什么区别，如何实现程序从 NAND FLASH 启动，如何使用 TFT 液晶，ARM 可执行映像文件是怎么构成的，什么是分散加载等。对这些问题在书中都有阐述。

此外，本书还向读者展示了一个启动代码的实现，当读者真正理解了启动代码的含义且对分散加载机制有一定了解后，将会发现移植 Uboot 已不再是什么难事，而且可以清晰地理解 Uboot 的启动流程和编写思路，可以清楚地知道启动代码是如何引导 Linux 内核……

本书自始至终秉承这一理念，提供的是机制而不是策略，“纸上得来终觉浅，绝知此事要躬行”，从点亮一个 LDE 开始讲起，最终使读者能够在 TFT 液晶上实现电子相册，书中实验部分仅仅是为了帮助读者理解硬件资源，读者可以自行修改以实现自己的功能。

前　言

在嵌入式系统设计中，基于 ARM 的应用占据了较大的市场份额，但是很多习惯了单片机开发的工程师或者从来没有接触过ARM的高校学生，在听到ARM嵌入式系统设计时，难免会有太复杂、很难入门的感觉。另外，一方面，很多 ARM 开发板供应商提供的开发板使用手册中讲解完开发板硬件资源后马上就移植操作系统（基本上是移植 Linux 2.6 内核），这也使得很多人以为使用 ARM 就必须使用操作系统；另一方面，即使有部分实验教程是讲解裸机开发，也是在 Linux 环境下讲解，这无形中给初学者增加了入门的难度，因为在 Linux 环境下的 Makefile 编写本身就有很多内容需要学习。本书就是为了弥补这一不足，使初学者能够从 8 位的单片机开发顺利转移到 32 位的 ARM 开发中。

作为一名初学者，在学习新知识的时候很难静下心来去阅读大篇幅的概念性叙述。至少笔者当初在学习的时候是这种心理，总以为那些描述是写给明白人看的。因此，本书的主线是：从一个功能强大的单片机的角度去理解 ARM 处理器，从实践的角度去理解 ARM 开发基础知识，从前后台系统的角度去理解 ARM 裸机开发，突出重点，各个击破，争取从实践的角度去找到与理论的吻合点。

本书的特点

- 理论与实践相结合。本书以实例为基础，详细阐述了基于 ARM 的嵌入式系统开发所需要的基础知识，同时恰当地摒弃了部分对于初学者而言暂时不用或者很少用到的知识点（例如，ADS 命令行开发工具的使用和 Thumb 指令），目的在于尽量使学习重点突出。
- 模块化设计与系统设计相结合。本书第 6～14 章，各个章节构成一个模块，由浅入深，读者可以有选择地阅读，最后在提高篇，将部分模块组成了一个较大的系统，读者可以形象地看到模块化开发的全貌和实现过程。

本书的编写原则

- 尽量展现细节，即使在有些情况下显得有点啰唆。书中有些地方可能看似很简单，显得有点啰唆，但是为了给初学者展现出 ARM 开发的全貌，编者选择了这种编写风格。因为编者在过去的学习过程中遇到很多问题，到论坛发帖求助，查资料，弄了好长时间才解决，因此为了给读者提供一个完完整整的开发过程，尽可啰唆一点，也不漏掉细节问题。
- 代码注重的是可读性，没有考虑过效率和编程规范是什么东西。本书代码力求通俗易懂，并没有考虑程序执行的效率和编程风格等。读者应对基本的编程有大概的了解，才有可能谈及编程规范。因此，尽快掌握编程才是硬道理，其他问题后续解决。

> 尽量用朴实的语言去描述看似深奥的理论。编者努力使本书作为一本 ARM 开发的纪实手册，努力展现出开发过程中的问题及其解决方法，尽量给读者提供一个参考，使读者少走弯路。因此，编者选择以尽量通俗的语言来叙述，并不想用艰深、晦涩、难懂的语言来迷惑读者。

本书内容概述

- 第 1 章简要讲解了 ARM 处理器的部分基础知识，同时给出了天嵌 TQ2440 开发板的硬件组成，这也是本书的硬件平台。关于具体硬件并没有给出过多的解释，这部分内容放在后续章节实验部分中。
- 第 2 章对 ADS 1.2 进行了讲解，摒弃了部分初学者暂时用不到的功能，突出重点，此外，给出了裸机程序的下载方法。
- 第 3～5 章对 ARM 汇编语言和 ARM C 语言进行了讲解，同时对 ARM 汇编语言和 C 语言混合编程进行了阐述。
- 第 6 章将前面章节的内容进行了综合，以实例的形式向读者展示了 ARM 裸机开发平台搭建、程序编写、下载到开发板执行的全过程。
- 第 7 章对 ARM 启动代码进行了分析，给出了一个启动代码的实例。
- 第 8～14 章对 S3C2440 处理器硬件资源进行了详细讲解，同时对每个模块给出了一个具体实现方法，以便使理论结合实践，在实践中更好地理解各个模块的使用方法和使用过程中的注意事项。
- 第 15 章是综合实战部分，给出了三个具体的实例，分别是数据采集系统、串口控制和电子相册的制作，这三个具体的实例都是基于前面各个模块开发为基础的。通过这三个实例的学习，读者可以进一步巩固本书的理论知识。
- 第 16 章是理论扩展部分，对嵌入式系统电源设计进行了探讨并给出了具体设计实例，同时也给出了两个基本的 Linux 内核实验，向初学者展示了 Linux 内核开发的基本流程。

本书配有光盘，其中含有本书所有实验的源代码以及本书相关的配套视频教程，读者可以结合光盘进行学习。

此外，本书只是对 ARM 裸机开发进行了讲解，熟悉裸机开发是进行操作系统开发的基础，秉承本书的风格，一切从简单开始，对功能进行逐步扩展，最终实现较为复杂的系统，在后续编写计划中，笔者将对嵌入式实时操作系统 μC/OS-II 进行讲解，尽量详细地向读者展示操作系统移植的过程以及如何使用操作系统进行应用程序开发，敬请期待。

本书内容侧重于实用性，讲解基于广州天嵌计算机科技有限公司的 TQ2440 开发板的实际电路图、数据手册和技术资料，为了便于读者查阅和与原文保持一致，本书对某些不符合国家标准的图形、符号等未做改动，未对某些原图或表进行翻译，敬请谅解。

参与本书编写的人员主要有王小强、李英花、方鹏、白甜甜、徐婋、田飞、曹淑明、程风明、周琛辉、陈小军等，粟思科审校。在本书编写过程中，编者得到了成都智制者科技有限公司欧阳骏、李岩、吴川、张凯之、张剑、龙宇等工程师们的支持与帮助，同时也

得到了孟海滨、梁飞、赵明辉、李亮以及肖菊兰等工程师们的支持与帮助。本书的顺利出版也离不开广州天嵌计算机科技有限公司马义忠、梁传智、黄健、戴俊秀等工程师们的大力支持。同时，参与本书编写工作的人员还有王治国、冯强、曾德惠、许庆华、程亮、周聪、黄志平、胡松、邢永峰、邵军、边海龙、刘达因、赵婷、马鸿娟、侯桐、赵光明、李胜、李辉、侯杰、王红研、王磊、闫守红、康涌泉、蒋杼倩、王小东、张森、张正亮、宋利梅、何群芬、程瑶，在此一并表示感谢。

此外，需要特别感谢赵建平教授对笔者的指导与帮助，同时也感谢吴星远老师、袁金霞老师的支持，他们的支持与鼓励是编者前进的最大动力。

由于编者水平有限，书中难免有不恰当的地方，恳请各位老师及同行提出宝贵的意见，联系方式为：china_54@tom.com，欢迎来信交流。

编　者

目　录

第1篇　基　础　篇

第 2 篇 提 高 篇

第 3 篇 典型项目分析

第 4 篇 理论知识扩展

第 1 篇
基 础 篇

第 1 章 ARM 处理器简介

最初的 ARM 处理器由英国剑桥的 Acorn 计算机公司（是 ARM 公司的前身）设计。ARM 公司成立于 1990 年，该公司是知识产权（IP）提供商（不生产芯片）。目前，ARM 架构处理器已在高性能、低功耗、低成本的嵌入式应用领域中占据了领先地位。

ARM 公司作为嵌入式 RISC 处理器的知识产权 IP 供应商，本身并不直接从事芯片生产，而是将设计许可授权给合作公司，合作公司添加自己的外设，进而生产各具特色的 SoC 芯片，利用这种合伙关系，ARM 很快成为许多全球性 RISC 标准的缔造者。目前，全世界有几十家大的半导体公司都使用 ARM 公司的授权，其中包括 Intel、IBM、Samsung、LG 半导体、NEC、SONY、PHILIP 等公司。因此，采用 ARM 处理器进行嵌入式系统开发时，开发者可以获得更多的第三方工具和技术支持，进而从一定程度上降低整个系统的研发成本，缩短研发周期，从而使产品更具市场竞争力。

ARM 体系结构基于精简指令集计算机（RISC）原理。RISC 的相关译码机制比复杂指令集计算机（CISC）的设计更简单，从而有更高的指令吞吐率、出色的实时中断响应。ARM 处理器的特性：

- 存取式体系结构（Load/Store，加载/存储式结构）。
- 小体积、低功耗、低成本、高性能。
- 16 位/32 位双指令集（16 位指令集使得所需程序存储器更小）。
- 流水线结构。

此外，数据存取指令的执行时间远远大于寄存器内部数据的操作指令的执行时间。因此，RISC 处理器都采用了 Load/Store 结构，只有专门的访存指令才可以与存储器打交道，其余指令不能进行存储器操作。ARM 也是采用 Load/Store 的结构，具有专门的 Load/Store 指令。同时，为了进一步提高指令和数据的存取速度，有的 ARM 处理器还有专门的指令高速缓存 ICache 和数据高速缓存 DCache。

1.1 处理器工作模式

ARM 处理器有 7 种工作模式，如表 1-1 所示。

表 1-1　ARM 处理器工作模式

处理器模式	说　明
用户模式（User）	程序正常执行的模式
快速中断模式（FIQ）	用于高速数据传输
外部中断模式（IRQ）	用于普通的中断处理
特权模式（Supervisor）	操作系统使用的一种保护模式
数据访问终止模式（Abort）	用于虚拟存储及存储保护
未定义指令终止模式（Undefined）	用于支持通过软件仿真硬件的协处理器
系统模式（System）	用于运行特权级的操作系统任务

ARM 处理器的工作模式分为用户模式和特权模式，除用户模式外的其他 6 种工作模式为特权模式。此外，除用户模式和系统模式外的其他 5 种工作模式又称为异常模式。

ARM 微处理器的工作模式可以通过软件改变，也可以通过外部中断或异常处理来改变处理器的工作模式。大多数的应用程序工作在用户模式下，当处理器工作在用户模式下时，某些被保护的系统资源是不能被访问的。

注意：本书旨在帮助读者尽快掌握 ARM 处理器的裸机开发，达到熟练掌握 ARM 处理器硬件资源和软件编程的目的。因此，主要使用的工作模式有用户模式、系统模式和外部中断模式。当然，系统上电后进入管理模式。其实，各种工作模式是为以后移植操作系统准备的。作为初学者，可以跳过这一部分，等到后面做实验的时候再复习这一部分。

1.2　寄存器介绍

ARM 寄存器共有 37 个寄存器，其中包括 31 个通用寄存器和 6 个状态寄存器，这些寄存器都是 32 位的，如图 1-1 所示。

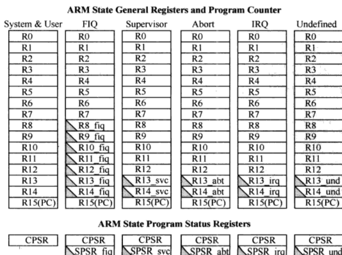


图 1-1　ARM 处理器的寄存器

ARM 处理器共有 7 种工作模式，每种工作模式都有对应的寄存器。当程序执行时，可见的寄存器主要有：15 个通用寄存器（R0～R14）、程序状态寄存器（CPSR 或者 SPSR）、程序计数器 R15（PC）。由图 1-1 可以看出，有些寄存器是各个模式公用的（也就是说，它们是同一个物理寄存器，因此当切换工作模式的时候，要保存这些寄存器的值），如 R0～R7；有些寄存器是各个模式独自拥有的物理寄存器（如图 1-1 中标黑色三角号的寄存器），当进行工作模式的切换时，这些寄存器主要用于保存原工作模式的寄存器的值。曾经有一个做嵌入式开发的公司，在面试时给出这样的题目：为什么快速中断模式比外部中断模式中断响应的速度要快？如果对这个概念还不熟悉，请阅读本书。

1.2.1 堆栈指针寄存器 R13 和链接寄存器 R14

对 ARM 处理器而言，R13 被称为堆栈指针寄存器，每种工作模式都有自己的堆栈指针寄存器，即每种工作模式下，都有一个物理的 R13。因此，当系统启动时，用户需要将所用到的所有模式下的 R13 初始化（在后面讲解启动代码部分会详细展开），在启动代码里面，所谓的初始化各个模式的堆栈其实就是将对应模式下的 R13 赋给适当的值即可（当然，用户需要了解自己所设计的系统的内存空间，一般而言，将堆栈放在内存的高地址端）。

R14 又被称为链接寄存器（Link Register），主要用于存放子程序的返回地址，跟 R13 一样，每种工作模式都有自己的 R14，在后面讲解工作模式切换时会详细讲解。

1.2.2 程序计数器 R15

寄存器 R15 常被用作程序计数器，记作 PC。要注意一点，ARM 采用流水线机制，程序计数器 PC 的值并不是指向当前正在执行的指令，对于 ARM 指令集而言，PC 总是指向当前指令的下两条指令的地址（不管是三级流水线还是五级流水线都是一样的）。因此，当异常发生时（还记得前面提到的异常工作模式吗？），处理器进入相应的异常工作模式，但是在处理完异常后，会返回到发生异常的地方接着执行，这是如何计算从异常模式返回的地址呢？这就涉及对 PC 指针的调整，在本书第 11 章的中断处理部分对此有详细的讲解。

1.2.3 程序状态寄存器

程序状态寄存器包括当前程序状态寄存器（CPSR）和备份程序状态寄存器（SPSR），这两种程序状态寄存器的格式一样，如图 1-2 所示。

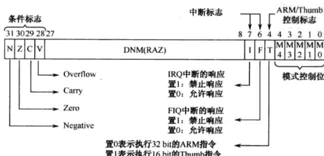


图 1-2　程序状态寄存器

ARM 指令可以根据条件来执行，当条件不满足时就不执行（这有点类似于高级语言里的条件语句），这里的条件就是条件标志位中对应的位 N、Z、C、V。关于条件标志位的作用，在本书第 3 章讲解 ARM 汇编指令时将根据具体用到的指令展开详细讨论。在 CPSR 中，模式控制为 M4—M0，用于标志处理器的工作模式（如表 1-2 所示）。所谓的处理器工作模式切换是：用专门的访问 CPSR 的指令，对 M4—M0 赋以相应的值，当然可以针对其中的某个域（特定的几位，而不是所有的位）进行操作。

表 1-2　模式控制位 M[4:0]

M[4:0]	处理器工作模式
10000	用户模式（User）
10001	快速中断模式（FIQ）
10010	外部中断模式（IRQ）
10011	管理模式（User）
10111	数据访问终止模式（Abort）
11011	未定义指令终止模式（Undefined）
11111	系统模式（System）

1.3　工作状态

ARM 处理器有两套指令集，对应着两种工作状态：执行 32 位 ARM 指令的状态和执行 16 位 Thumb 指令的状态。本书所选的处理器是三星公司的 ARM9 处理器 S3C2440，支持这两种指令集，但是本书主要是想给读者展现出 ARM 裸机开发的全貌，关于 Thumb 指令一般用在编译器优化阶段或者其他应用背景下。因此，本书不做具体讨论，读者可以参阅其他有关文献。对于初学者，请大胆地略过 Thumb 指令，这将加快前进的步伐，当学到一定程度后，可根据需要有选择地学习 Thumb 指令。

1.4　数据长度

ARM 处理器支持下列数据类型。

- 字节型数据（Byte）：数据宽度为 8bits。
- 半字数据类型（Half Word）：数据宽度为 16bits，必须以 2 字节对齐的方式存取。
- 字数据类型（Word）：数据宽度为 32bits，必须以 4 字节对齐的方式存取。

注意：ARM 有专门的指令来实现对不同数据类型的操作。

1.5　存储系统

对于不同的嵌入式系统而言，ARM 存储系统的复杂程度有所不同。因此，ARM 处理器提供了用于存储管理的存储器管理部件 MMU 以及协处理器 CP15，以支持复杂的内存管

理功能。当然，不同的内核所采用的内存管理方式不同，有的简单，有的复杂。此外，对某一具体的应用领域和硬件环境采用的内存管理方式也不同。这些内容本身就较为复杂，尤其是如果对虚拟存储技术没有一定程度的理解，则学习难度也很大。因此，本书采用的是平板模式（Flat），适用于小型嵌入式系统、系统中的任务比较少且数量固定的场合。

1.5.1 ARM 地址空间

本书旨在引导读者快速理解 S3C2440 处理器的硬件资源。因此，并没有进行复杂的地址映射，而是像一些单片机系统一样，系统中各部分使用的都是物理地址（这种模式成为平板式（Flat）的地址映射），对单片机较熟悉的读者可以很容易地理解和掌握。

1.5.2 ARM 存储器的格式

端模式（Endian）的这个词出自 Jonathan Swift 在 1726 年写的一篇讽刺小说《格利佛游记》。

书中根据将鸡蛋敲开的方法不同，将所有的人分为两类：从圆头开始将鸡蛋敲开的人被归为 Big Endian，从尖头开始将鸡蛋敲开的人被归为 Little Endian。大意是：吃鸡蛋前，原始的方法是打破鸡蛋较大的一端。可是当时皇帝小时候吃鸡蛋，一次按古法打鸡蛋时碰巧将一个手指弄破了，因此，他的父亲就下了一道敕令，命令全体臣民吃鸡蛋时打破鸡蛋较小的一端，违令者重罚。老百姓们对这项命令极为反感……历史告诉我们，由此曾发生过六次叛乱，其中一个皇帝送了命，另一个丢了王位……

在各种计算机体系结构中，对于字节、字等的存储机制有所不同，到底是将数据的高位存放在高地址端还是存放在低地址端呢？至今也没有统一的定论。因此，在计算机通信领域中存在一个很现实的问题，即通信双方传输的信息单元应该以什么样的顺序进行传送。如果不达成一致的规则，通信双方将无法进行正确的编/译码，从而导致通信失败。目前，在各种体系的计算机中通常采用的存储机制主要有两种：大端（Big-endian）和小端（Little-endian）。当不同端模式的计算机进行通信时，需要进行相应的转换。

- 大端：数据的高位存放在存储器低地址端，数据的低位存放在存储器高地址端。
- 小端：数据的高位存放在存储器高地址端，数据的低位存放在存储器低地址端。

例如，字型变量 A：A=0xFF7744CC，在内存中的起始地址为 0x30000000，在内存中的起始地址为 0x30000000，其在内存中的存放格式如图 1-3 所示。

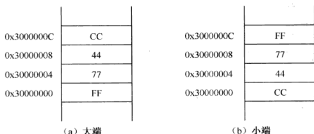


图 1-3 数据在内存中的存放格式

1.6 天嵌 TQ2440 开发板硬件资源概述

TQ2440 开发板是广州天嵌计算机科技有限公司设计的针对三星公司 S3C2440 处理器的开发板。开发板上提供了按键、LED、蜂鸣器等常用的功能部件，还拥有 RS-232 接口电路。此外，还外扩了 NAND FLASH、NOR FLASH、SDRAM 等，具体外设情况请读者参见相关开发板手册，如图 1-4 所示仅仅是给出了本书所涉及的部分模块。

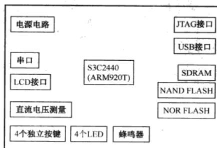


图 1-4 TQ2440 开发板功能框图[①]

注意：各个模块与处理器的接口方式并没有详细给出，在后面实验部分会给出详细的解释。

TQ2440 开发板分为两部分：底板和核心板。底板主要包括一些基础外设接口，如 LED、LCD、UART 等。核心板主要包括 S3C2440 处理器、SDRAM、NAND FLASH、NOR FLASH 等。拿到开发板时，首先需要对板子的基本元器件有个大概的了解。底板俯视图如图 1-5 所示。

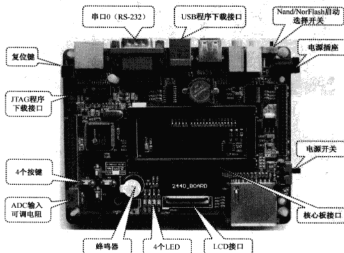


图 1-5 TQ2440 底板俯视图

① 本图所列的是 TQ2440 开发板的部分功能模块，对于其他未列出的模块，读者可以参见 TQ2440 开发板手册。目的是想尽量将本书用到的模块突出。

核心板分正面和反面，正面元器件布局如图 1-6 所示，主要有 S3C2440 处理器、NAND FLASH 芯片和 1 片 SDRAM 芯片。

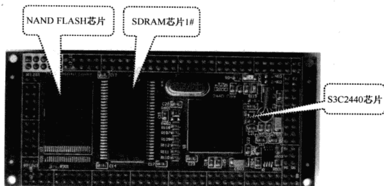


图 1-6　核心板正面俯视图

核心板反面元器件布局如图 1-7 所示，主要有 NOR FLASH 和 2 片 SDRAM 芯片，为什么需要 2 片 SDRAM 呢？这主要是为了扩充 SDRAM 的容量，每片 SDRAM 的容量是 32MB，两片加起来就是 64MB。

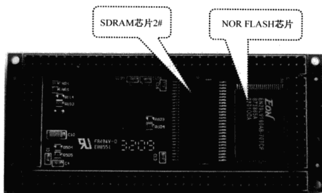


图 1-7　核心板反面俯视图

由开发板硬件资源引发的思考：有消息指出，微软公司已经在美国国际消费电子展上宣布，下一代 Windows 操作系统将支持 ARM 处理器，这使得 ARM 颇受业内关注。目前，ARM 处理器已经被广泛用于智能手机和平板电脑中。在 Windows 系统支持 ARM 处理器之后，ARM 将进军 PC 市场。不过，ARM 仍将把关注重点放在智能手机和平板电脑市场。可见，ARM 处理器已经在挑战传统的 X86 构架的处理器了。

对于初学者，面对众多的板载资源可能会有如下疑问：

- 为什么需要专门的电源电路呢？
- 什么是 SDRAM 呢?
- 什么是 NAND FLASH 和 NOR FLASH 呢，NAND FLASH 和 NOR FLASH?它们有什么区别呢?

- 在 ARM 系统中，程序放在什么地方呢?
- 系统上电后从哪里执行第一条指令呢？

此外，市面上有很多针对三星公司 S3C2440 的开发板，经过对比会发现，板载资源几乎是差不多的，这又是为什么呢？因此，要想了解上面问题的根源，请读者慢慢阅读本书。

1.7 本章小结

本章主要讲述了 ARM 处理器的基础知识，包括寄存器、工作状态和存储系统等，此外还给出 TQ2440 开发板的部分功能模块图，使读者对硬件模块有一个整体的概念。还有一个问题可能会困扰着部分读者：为什么 ARM 处理器有那么多的寄存器呢？本章扩展阅读部分会给读者提供部分信息。

1.8 扩展阅读之 CISC 处理器和 RISC 处理器简介

在过去相当长的一段时间里，计算机性能的提高往往是通过不断增加系统硬件的复杂性来实现的。但是，随着集成电路技术的迅速发展，特别是超大规模集成电路（VLSI）技术的发展，为了提高软件编程的灵活性和提高程序的运行速度，硬件工程师采用的办法是：不断增加可实现复杂功能的指令和多种灵活的编址方式，这样做的直接后果是硬件越来越复杂，硬件设计成本越来越高。此外，为实现复杂操作，微处理器除了向程序员提供各种寄存器和机器指令功能外，还采用了微程序设计的方法，即将指令功能分解成微操作，将微操作进行编码，然后将编码存入程序 ROM 中，微命令经过译码后就可以完成相应的功能。这一设计思路在一定程度上可以降低电路的复杂性，同时也有利于产品的更新换代，但缺点是微命令的实现速度较慢。这种设计的形式被称为复杂指令集计算机（Complex Instruction Set Computer，CISC）结构，常见的 X86 就是 CISC 结构的处理器。

因此，传统处理器设计上曾面临速度、复杂性、设计周期以及产品更新换代等的矛盾。当计算机的设计沿着这条道路发展时，面对这些问题，有些人开始怀疑这种传统处理器设计做法，重新审视了处理器的设计过程，试图从设计理念上找到解决问题的方法。

IBM 公司设在纽约 Yorktown 的 Jhomas I.Wason 研究中心于 1975 年组织力量对指令系统进行了大量研究。结果表明，软件中大部分的指令为简单指令，约占 80%左右，但是这部分简单指令的运行时间占总运行时间的 20%左右；软件中复杂指令只占一小部分，占 20%左右，但是这部分复杂指令的运行时间却占了总处理器运行时间的 80%左右。

因此，人们得出的结论是：从指令集中去掉复杂指令，而复杂指令的功能由软件来实现，这样可以简化电路设计，同时去掉微程序设计，采用硬连控制的方法来进一步提高处理器的运行速度。但是这其中涉及一个问题：复杂指令功能由软件实现能真正提高处理器的速度吗？答案是肯定的。因为程序中对复杂指令的使用频率较低，简单指令有利于流水线的执行，由于去掉了存储微程序的 ROM，所以可以简化电路，从而节省集成电路芯片的面积，这部分面积可以用来增加 Cache 的容量，从而使处理器的速度得到明显的提高。

因此，针对 CISC 的这些弊病，有人提出了精简指令的设想，即只保留指令系统中那

些使用频率很高的少量指令，同时提供一些必要的指令用以支持操作系统和高级语言。按照这个原则发展而成的计算机被称为精简指令集计算机（Reduced Instruction Set Computer，RISC）结构。现在的ARM处理器就是RISC结构的处理器。

RISC处理器的基本特征如下。

- 简单固定的指令格式：
  - ◆ 指令长度固定。大多数RISC处理器指令长度一般设定在总线宽度以内，尽量保证取指令码在一个总线周期完成，避免了多周期取指造成的流水线阻塞，同时指令长度无须译码，这样简化了译码电路并节省了指令长度译码时间。
  - ◆ 指令字段位置固定。
  - ◆ 指令意义简单。功能单一，简化硬件逻辑。
- 大容量高速缓存[①]。缓存更多的指令和数据，减少访存次数。
- 流水线技术。尽量使指令在单周期执行完成，避免流水线阻塞，RISC的设计思想更利于指令按流水线方式的运行。
- 大量寄存器。尽量保证上下文切换尽可能在寄存器中完成。
- 硬连控制。以简化的指令集为基础，提高指令执行速度。
- 采用存取式体系结构（Load/Store结构）。仅专门的访存指令才允许访存，避免执行周期访存造成的流水线阻塞。
- 哈佛（Harvard）总线结构。采用相互分离的ICache和DCache，利用双总线动态访问机构，使数据存取和指令预取可以并行。
- 重叠寄存器窗口技术。将大量寄存器分成多个重叠寄存器窗口，用以在执行高级语言程序中的过程调用和返回时直接传递参数，减少了调用和返回时访问主存所消耗的时间。
- 优化编译技术。按照指令执行速度的快慢以及CPU的流水线深度等合理调整指令顺序，使CPU最大限度地让流水线并行执行。

请读者注意，上面列出的特征并不是每一款RISC处理器都具备的，对某一款特定的RISC处理器来说，可能会注重某几个方面的特征。到此为止，读者可能会理解为什么ARM处理器有37个寄存器，其实ARM处理器也是一款RISC处理器，因此会有较多的寄存器。

---

① 关于高速缓存的知识，如果读者不是很熟悉的话，可以略过此部分，本书后面扩展阅读部分会有详细的解释。同时，高速缓存的原理和结构不在本书讨论范围内，读者可以根据自己的实际情况有选择地阅读。

第2章 ADS 集成开发环境及程序下载具体流程

ADS（ARM Development Suit）是 ARM 公司推出的嵌入式微控制器集成开发工具，目前成熟版本是 ADS 1.2。它的前身是 SDT，SDT 是 ARM 公司早期的嵌入式集成开发环境，SDT 早已经不再升级。ADS 集成开发环境由命令行开发工具、ARM 运行时库、GUI 开发环境（Code Warrior 和 AXD）组成。用户可以为 ARM 系列的 RISC 处理器编写和调试自己的应用程序。ADS 1.2 支持 ARM 10 之前的所有 ARM 系列微控制器，支持软件调试及 JTAG 硬件仿真调试，支持汇编、C、C++源程序，具有编译效率高、系统库功能强等特点，可以在 Windows 98、Windows XP、Windows 2000 等上运行。

本书定位在使用 ADS 1.2 提供的 GUI 开发环境（Code Warrior 和 AXD）进行 S3C2440 处理器的裸机开发，因此不会过多地讨论 ADS 1.2 提供的命令行开发工具，建议读者略过命令行部分，这不会影响本书的学习[①]。

2.1 ADS 1.2 集成开发环境简介

ADS 1.2 集成开发环境作为一款优秀的嵌入式微控制器集成开发工具，一方面可以满足不同层次系统设计和开发的要求，另一方面也决定其复杂性。但作为初学者一般直接使用的是 CodeWarrior IDE 集成开发环境和 AXD 调试器（如表 2-1 所示），表中所列内容可以满足初学者的学习要求。

阅读这一章节的内容，读者应该有这样的疑问：什么是 ARM 汇编器？什么是 ARM 连接器？Fromelf 有什么作用？……这些疑问在后续的章节中会逐步地展开，读者会逐渐明白 ARM 程序的组织结构、开发流程以及工具的具体作用及用途。在此处，读者大可不必过于

---

① 本教程没有大篇幅地介绍 ADS 的命令行开发。本书的观点是，如果读者在开发中确实需要命令行，则直接学习 Linux 下面的裸机开发即可（众所周知，Linux 对命令行的支持远比 Windows 要好得多），没有必要在 Windows 下学习命令行开发，毕竟，选择在 Windows 下进行裸机开发，出发点就是 Windows 对图形界面的支持较好，可以帮助初学者更好地理解裸机程序的开发过程（编译、链接等内容还是在 Linux 环境下学习较好）。但是，Linux 对命令行有很好的支持，在 Linux 学习命令行的开发（借助于 Makefile）会对以后 Uboot 的移植、操作系统的移植等有很大的帮助。

关心这些工具的作用，只要熟悉大体的流程即可，在后面实验部分，会有更深入的内容。

表 2-1　ADS 1.2 集成开发环境部分组件

工 具 名 称	功能与作用	使 用 方 式
代码生成工具	ARM 汇编器 ARM C/C++编译器 ARM 连接器	在 CodeWarrior IDE 中调用这些工具
集成开发环境	CodeWarrior IDE	工程的管理、编译与链接
调试器	AXD Debugger	在线仿真调试
指令模拟器	ARMulator	在 AXD Debugger 中调用
ARM 开发包	Fromelf	在 CodeWarrior IDE 中调用
ARM 库	C/C++库函数等	用户应用程序调用

2.1.1　CodeWarrior for ARM 开发环境

CodeWarrior for ARM 是一套完整的集成开发工具，该工具是专为基于 ARM RISC 的处理器而设计的，它可加速并简化嵌入式开发过程中的每一个环节，使开发人员只需通过一个集成软件开发环境就能开发出 ARM 产品。在整个开发周期中，开发人员无须离开 CodeWarrior for ARM 开发环境，因此节省了在开发工具上花的时间，使开发人员有更多的精力投入到代码编写上。CodeWarrior for ARM 集成了 ARM 汇编器、ARM C/C++编译器和 ARM 连接器等，包含工程管理、代码生成、关键词高亮显示等功能，可以满足读者的嵌入式开发需求。CodeWarrior 的主界面如图 2-1 所示。

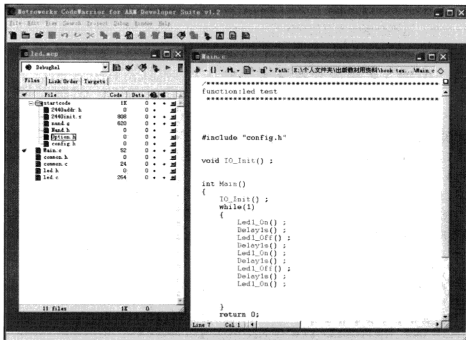


图 2-1　CodeWarrior for ARM 的主界面

注意：建议读者结合本书视频教程学习以下内容，毕竟开发工具的学习只要看着别人做一遍，很快就会掌握，自己看书的话可能会慢一些。

2.1.2 AXD 调试器的启动

当用户程序编写完成后，就可以启动 AXD 调试器进行程序的调试。AXD 调试器支持单步、全速、执行到光标处、断点等调试功能，可以观察变量、寄存器和内存单元的内容。启动 AXD 调试器有两种方法。

第一种方法是依次单击：开始\程序\ARM Developer Suite v1.2\AXD Debugger 调试器，如图 2-2 所示。

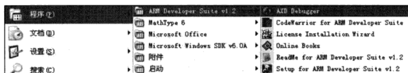


图 2-2　启动 AXD 调试器

第二种方法是在 CodeWarrior for ARM 里单击“Debug（调试）”按钮即可启动 AXD 调试器，如图 2-3 所示。

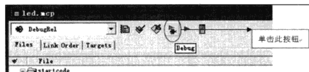


图 2-3　从 CodeWarrior 启动 AXD 调试器

这里需要注意的问题是：用第二种方法启动 AXD 调试器时，有时会出现不正常的现象。当读者遇到这种现象时，可以尝试用第一种方法启动 AXD 调试器，本书推荐用第一种方法启动 AXD 调试器。

此外，启动 AXD 调试器后还要设置 AXD 调试器，然后加载可执行映像文件，这些内容请读者参见本章 2.3 节“工程的调试”和第 3 章 3.4 节“用 AXD 调试 ARM 汇编程序实验”。

启动 AXD 后，主界面如图 2-4 所示。

2.2 工程的编辑与修改

进行基于 ARM 的程序开发时，CodeWarrior for ARM 提供了工程管理、源文件编辑、程序编译链接等功能。下面对每一部分内容进行具体讲解。如果读者还没有进行过 ARM 程序的编写，请忽略所有程序的具体内容，只关注流程就可以。当看完后面的章节后，再回过头来看这部分内容时，会发现原来也不过如此。但是，为了给初学者（特别是 ARM 新手）提供一个较为详细的学习平台，笔者还是决定把这部分内容写在这里。



图 2-4 AXD 调试器主界面

2.2.1 建立一个新工程

依次单击：开始\程序\ARM Developer Suite v1.2\CodeWarrior for ARM Developer Suite，启动 Metrowerks CodeWarrior，启动 ADS 1.2 如图 2-5 所示。

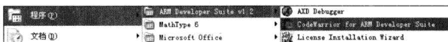


图 2-5 启动 ADS 1.2

单击 File\New 即可弹出"New(新建工程)"对话框，如图 2-6 所示。选择 ARM Executable Image[①]（ARM 可执行映像文件），输入工程名，例如："test"，选择保存路径即可。

2.2.2 建立一个源文件

读者可以在刚才建立的工程下建立一个文本文件，以便输入用户程序，单击"New Text File"按钮，如图 2-7 所示。然后，在新建的文件中输入源程序，保存即可。保存时，要根据建立文件的类型，输入文件的全名（如建立一个 C 语言源文件，保存时输入的文件名可以是 test.c。若建立的是一个汇编语言源文件，则文件名为 test.s）。一般情况下，应当将同一个项目所需的文件保存在同一目录下，以便于文件的管理和查找。

① 为什么选择 ARM 可执行映像文件呢？读者可以暂时不考虑，等读者读完本书，把实验都做完的时候，就可能会发现，在初学阶段，用到的就只有这种格式的工程。等读者入门后，可以根据具体的项目要求，恰当地选择不同类型的工程。这就好比 Microsoft Word 中"工具"菜单下有好多工具，如共享工作区、联机协作、宏、模板与加载项等功能。相信很多读者可能根本就没用过此功能，但是这并不影响读者用 Microsoft Word 进行基本的文字处理。

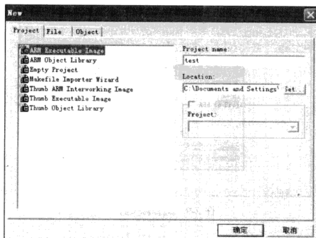


图 2-6 “New（新建工程）”对话框

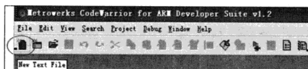


图 2-7 新建文件

2.2.3 添加源文件到工程

在工程窗口中单击鼠标右键，可以弹出一个悬浮菜单，选择“Add Files”项，如图 2-8 所示，即可弹出“Select files to add”窗口，选择相应的源文件，单击“打开”按钮即可，如图 2-9 所示。

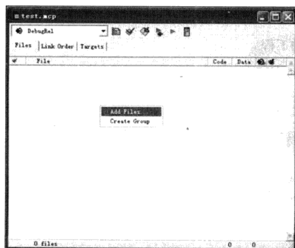


图 2-8 添加文件到工程

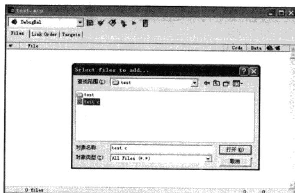


图 2-9　选择源文件

2.2.4　编译与链接工程

与工程编译、链接有关的几个按钮如图 2-10 所示。

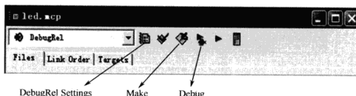


图 2-10　编译与链接相关按钮

- DebugRel Settings：设置工程的属性，如设置链接地址、输出文件的格式、编译选项等。
- Make：编译、链接。
- Debug：启动 AXD 调试器。

2.2.5　打开已有的工程

选择“File”菜单下的“Open”项，即可弹出“打开工程”对话框，选择相应的工程即可。

2.3　工程的调试

CodeWarrior for ARM 支持软件调试功能，这是借助于 AXD 调试器完成的。工程编译链接后会生产一个.axf 格式的文件，该文件可以装载到 AXD 里进行调试。

2.3.1　装载映像文件

按照 2.1.2 节的步骤，启动 AXD 调试器后，选择“File”菜单下的“Load Image”项，

如图 2-11 所示，即可打开“Load Image”对话框，选择相应的映像文件[①]即可，如图 2-12 所示。

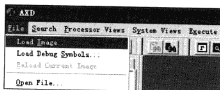


图 2-11　装载映像文件

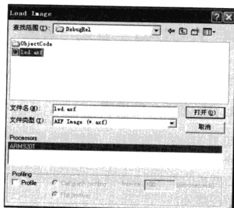


图 2-12　选择映像文件

2.3.2　调试工具条的使用

AXD 调试工具条主要有两类：控制程序运行的工具，如图 2-13 所示；查看寄存器和内存数据的工具，如图 2-14 所示。

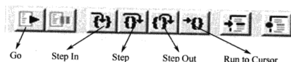


图 2-13　控制程序运行的工具

- Go：全速运行。
- Step In：单步执行，当遇到函数调用语句时，会进入被调函数中执行。
- Step：也是单步执行，但是当遇到函数调用语句时，并不进入被调函数里，而是将被调函数当做一条语句执行。
- Step Out：执行完当前被调函数后，停止在被调函数的下一条语句。
- Run to Cursor：运行到光标处，这个工具很有用，在后面章节中主要用这一工具进行程序调试。

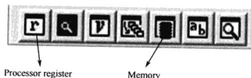


图 2-14　查看寄存器和内存数据的工具

① 这里所说的映像文件是编译、链接以后生成的.axf 格式的文件。如果读者没有 ARM 编程经验的话，可以略过这一部分，本书第 6 章会更加详细地展开这部分内容。

- Processor register：用于查看系统寄存器的值。
- Memory：用于查看内存单元的值。

2.4 H-JTAG 的安装与调试

H-JTAG 主要用于程序的下载，下面进行讲解。

2.4.1 H-JTAG 的安装

用 H-JTAG 下载程序，前提是电脑上要有并口，一般笔记本电脑没有并口，当然可以买一个 J-Link 来实现 USB 转并口功能，但本书并没有采用这种方式，因此下面讲的内容适合有台式机的读者。如果读者用的是笔记本电脑，请跳过本节，直接阅读使用 U-Boot 下载裸机程序部分。

安装 H-JTAG 的详细步骤请参见《TQ2440 开发板使用手册》。

2.4.2 H-JTAG 的设置

安装好 H-JTAT 后，要进行适当的设置才可以正常使用。下面讲解如何将程序下载到 NOR FLASH 中，关于将程序下载到 NAND FLASH 中的方法，请读者参见第 6 章。

（1）双击“H-JTAG”图标，选择“Settings”菜单下的“LPT Jtag Setting”项，如图 2-15 所示。设置界面如图 2-16 所示。

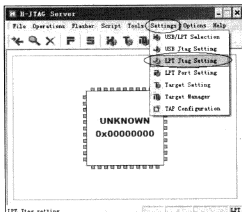


图 2-15 打开 H-JTAG

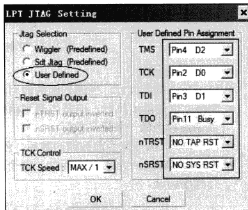


图 2-16 H-JTAG 设置

（2）开发板上电后，单击“Detect target”，软件会自动检测 CPU，检测成功后会显示 CPU 型号，如图 2-17 所示，CPU 型号为 ARM920T 0x0032409D。

然后，选择“Flasher”菜单下的“Start H-Flasher”项，如图 2-18 所示。

（3）在弹出的 H-Flasher 窗口中，选择“Load”，然后，选择“TQ2440_nor_eon.hfc”即可，如图 2-19 所示。

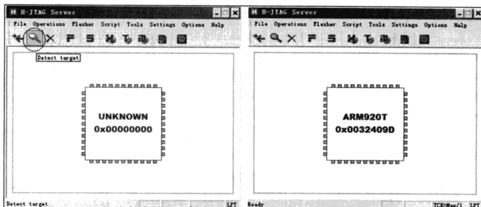


图 2-17　检测 CPU

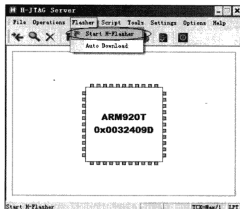


图 2-18　选择“Start H-Flasher”项



图 2-19　选择配置文件

说明：上述 H-JTAG 文件共有 4 个，具体选择哪一个配置文件，请参考具体的开发板硬件配置，以下内容摘自于广州天嵌计算机科技有限公司 TQ2440 开发板配套资料。

- TQ2440_nand_2KP.hfc：用于烧写 2K 大页面 Nand Flash 的 H-flash 配置文件。
- TQ2440_nand_64MB.hfc：用于烧写 512B 页面 Nand Flash 的 H-flash 配置文件。
- TQ2440_nor_eon.hfc：用于烧写 Nor Flash 的 H-flash 配置文件。
- TQ2440_nor_sp.hfc：用于烧写 Nor Flash 的 H-flash 配置文件。

（4）在 H-Flasher 窗口中，选择“Flash Selection”，然后选择“EN29LV160AB”（不同的开发板可能具体型号不同，读者需要确定 Flash 芯片的型号），如图 2-20 所示。

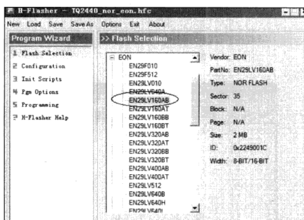


图 2-20　选择 Flash 型号

（5）在 H-Flasher 窗口中，选择“Programming”，单击“Check”按钮，会显示出 Flash 型号，单击“Src File”框右边的“…”按钮，会弹出“打开”对话框，然后找到“Eint1.bin”，最后单击“Program”按钮即可实现程序下载，整体过程如图 2-21、图 2-22 所示。

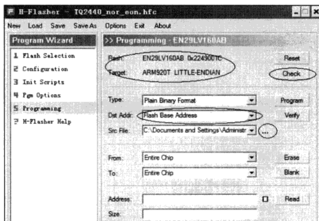


图 2-21　检测 Flash 型号

（6）程序下载结束后会显示如图 2-23 所示的界面。

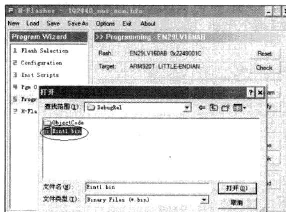


图 2-22　选择二进制文件

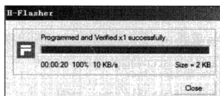


图 2-23　下载完成界面

注意：上述过程介绍得比较笼统，读者可以结合第 11 章进行深入学习。

2.5　使用 U-Boot 下载裸机程序

关于什么是 U-Boot 的问题，请读者暂时不要考虑了，在本书中使用 U-Boot 只是为了下载裸机程序到 NAND FLASH 中，当读者做完本书实验后，再去详细关注 U-Boot 即可。

如果读者使用的是笔记本电脑，那么需要使用 U-Boot 下载裸机程序。U-Boot 界面如图 2-24 所示。读者可能会问：怎么才能出现这个界面呢？

首先，需要将 U-Boot 下载到 NOR FLASH 中，然后从 NOR FLASH 启动，打开超级终端即可出现这个界面。当然，烧写 U-Boot 到 NOR FLASH、设置超级终端请读者参见《TQ2440 开发板使用手册》，在此不做赘述。

总体而言，在笔记本电脑没有并口和串口的情况下，使用 U-Boot 下载裸机程序到 NAND FLASH 开发工具方面主要分为以下几步：

（1）下载 U-Boot 到 NOR FLASH。

（2）需要有一根 USB 转串口线，然后安装 USB 转串口驱动。

（3）需要有一个 USB 下载线。

（4）设置超级终端。

（5）安装 DNW 软件。

关于什么是 NAND FLASH 和 NOR FLASH 的问题，请读者暂时放一下，现在只需要记住，TQ2440 开发板上有个开关，可以选择从 NAND FLASH 启动还是从 NOR FLASH 启动。

```
##### EmbedSky BIOS for SKY2440/TQ2440 #####
Press Space key to Download Mode !

#####     Boot for Nor Flash Main Menu      #####
[1] Download u-boot or STEPLDR.nb1 or other bootloader to Nand Flash
[2] Download Eboot to Nand Flash
[3] Download Linux Kernel to Nand Flash
[5] Download CRAMFS image to Nand Flash
[6] Download YAFFS image to Nand Flash
[7] Download Program (uCOS-II or TQ2440_Test) to SDRAM and Run it
[8] Boot the system
[9] Format the Nand Flash
[0] Set the boot parameters
[a] Download User Program (eg: uCOS-II or TQ2440_Test)
[b] Download LOGO Picture (.bin) to Nand  Flash
[l] Set LCD Parameters
[o] Download u-boot to Nor Flash
[r] Reboot u-boot
[t] Test Linux Image (zImage)
[q] quit from menu
Enter your selection:
```

图 2-24　U-Boot 界面

在上述界面中输入字母 a 或者 A，会出现如图 2-25 所示的界面，等待下载程序。

```
##### EmbedSky BIOS for SKY2440/TQ2440 #####
Press Space key to Download Mode !

#####     Boot for Nor Flash Main Menu     #####
[1] Download u-boot or STEPLDR.nb1 or other bootloader to Nand Flash
[2] Download Eboot to Nand Flash
[3] Download Linux Kernel to Nand Flash
[5] Download CRAMFS image to Nand Flash
[6] Download YAFFS image to Nand Flash
[7] Download Program (uCOS-II or TQ2440_Test) to SDRAM and Run it
[8] Boot the system
[9] Format the Nand Flash
[0] Set the boot parameters
[a] Download User Program (eg: uCOS-II or TQ2440_Test)
[b] Download LOGO Picture (.bin) to Nand  Flash
[l] Set LCD Parameters
[o] Download u-boot to Nor Flash
[r] Reboot u-boot
[t] Test Linux Image (zImage)
[q] quit from menu
Enter your selection: a
USB host is connected. Waiting a download.
```

图 2-25　程序下载界面

然后，打开 DNW 软件（关于 DNW 软件的设置请读者参见《TQ2440 开发板使用手册》），选择“USB Port”菜单下的“Transmit”项即可，如图 2-26 所示。

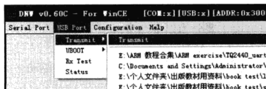


图 2-26　USB 下载界面

最后，下载程序到 NAND FLASH 主要分为以下几步：

（1）打开超级终端，开发板选择从 NOR FLASH 启动，出现 U-Boot 界面。

（2）输入字母 a 或者 A。

（3）打开 DNW 软件，选择“USB Port”菜单下的“Transmit”项。

（4）选择要下载的二进制文件即可。

2.6　本章小结

本章主要对开发工具进行了讲解，其中对 ADS 开发环境下工程的建立和添加源文件到工程以及编译等步骤都进行了分析。如果读者不是很熟悉或者看不懂，请不要放弃，开发工具只是辅助开发的，用一遍就熟悉了，等用过一遍以后，相信读者会觉得这一章写得是多么肤浅，但是初学者还是要了解一下，这样对相关工具会有个印象，以后用到的时候就不那么陌生了。因此，现在看不懂没有关系，第 6 章会讲解如何点亮一个 LED，读者可以结合第 6 章来学习。此外，本章所讲述的内容在本书光盘中有相关的视频教程，读者也可以参考。本章最后对使用 H-JTAG 和 U-Boot 进行裸机程序下载也进行了简要讲解，读者可以结合第 6 章来阅读。

第 3 章

ARM 指令集及汇编语言基础

面对 ARM 指令集以及 ARM 汇编语言程序设计，初学者往往感到无从下手，到底 ARM 汇编需要学习到什么程度才算可以呢？其实，读者大可不必去记忆每一条汇编指令，学习指令集最好是结合具体实验，先将基本的指令用熟，遇到新指令时查一下相关指令集手册即可。因此，建议读者先按照本书的内容编排学习，本章内容只是涉及了 ARM 指令集和汇编语言程序设计的部分内容，目的在于尽量使问题简单化，尽量将开发时最常用的指令进行全面介绍。

通过本章的学习，读者基本上可以看懂一个简单的启动代码（关于启动代码的知识，在第 7 章将给出具体详细的介绍）。因此，当读者真正掌握了本章内容后，如果在开发过程中遇到其他问题时，可以查阅 ARM 指令集进行详细学习，相信问题会迎刃而解。

3.1 ARM 指令集介绍

ARM 处理器是基于精简指令集计算机（RISC）原理设计的，指令集的译码机制较为简单，ARM920T 具有 32 位 ARM 指令集和 16 位 Thumb 指令集。ARM 指令集执行效率高，但是代码密度相对较低；而 Thumb 指令集是 ARM 指令集的子集，具有更好的代码密度，而且保持了 ARM 的大多数性能上的优势。所有 ARM 指令都是可以有条件执行的，而 Thumb 指令仅有一条指令具备条件执行功能。ARM 程序和 Thumb 程序可相互调用，相互之间的状态切换开销可以忽略不计。但是，本章对 Thumb 指令并没有涉及，因为初学者的确很少涉及或者用不到 Thumb 指令。请读者大胆放弃，等学完了 ARM 指令集后，如果读者本身的项目有需要 Thumb 指令的地方，读者可以自行学习。其实，学习完 ARM 指令集，Thumb 指令集就很简单。

3.1.1 ARM 指令集

本章或许叫做“启动代码中的 ARM 指令”更恰当一些，因为本章内容是从 ARM 指令集中选取部分指令进行讲解，这部分指令通常能够完成启动代码的编写。因此，如果读者对 ARM 指令集有相当了解的话，可以略过这一章，进行后面的学习。

ARM 指令的基本格式：<opcode>{<cond>}{S} <Rd>，<Rn>{，<opcode2>}。

其中，< >内的项是必须的（例如，<opcode>是指令助记符，是必须的），{ }内的项是可选的（如之类的执行条件{<cond>}是可选的），如果不写则使用默认条件是无条件执行。

- opcode 是指令助记符，如 MOV，LDR 等。
- cond 表示指令的执行条件（是 Condition 的缩写），如 GT、NE 等。用于控制指令条件执行的常用执行条件码如表 3-1 所示。注意：这里的条件是与当前程序状态寄存器 CPSR 的条件标志位对应的。
- S 决定是否影响 CPSR 寄存器的值，当书写时影响 CPSR，否则不影响。
- Rd 是目标寄存器。
- Rn 是第一个操作数的寄存器。
- operand2 是第二个操作数，是可选的，ARM 指令中第二操作数可以是立即数、寄存器或者寄存器移位等方式，在此不做赘述，用到具体指令时再详细讨论。

表 3-1　指令条件执行常用条件码

条件码助记符	标 志 位	含　义
EQ	Z=1	相等
NE	Z=0	不相等
CS	C=1	无符号数大于或等于
CC	C=0	无符号数小于
GT	Z=0，N=V	带符号数大于
LE	Z=1，N! =V	带符号数小于或等于
AL	\	无条件执行（指令默认条件）

1．存储器访问指令

ARM 处理器对 ROM、RAM 和 IO 地址采取统一编址，除对 RAM 操作以外，对外围 IO、程序数据的访问均要通过加载/存储（Load/Store）指令进行。ARM 的加载/存储（Load/Store）指令可以实现字、半字、无符/有符字节操作；批量加载/存储（Load/Store）指令可实现一条指令加载/存储多个寄存器的内容，大大提高效率。

- LDR 和 STR

加载/存储指令。LDR 指令用于从内存中读取数据加载到寄存器中；STR 指令用于将寄存器中的数据保存到内存。

例如，LDR R0，[R1]，表示将 R1 所指向的存储单元的内容加载到 R0 寄存器中；STR R0，[R1]，表示将 R0 寄存器里面的内容存储到 R1 所指向的存储的单元中。

- LDM 和 STM

批量加载/存储指令，可以实现在多个寄存器和一块连续的内存单元之间传输数据。LDM 指令实现加载一块连续内存单元的数据到多个寄存器，STM 将多个寄存器的内容存储到一块连续的内存单元中，因此这两条指令主要用于参数传递和数据复制。

指令格式：

LDM{cond}<mode>　Rn{!}，{reglist}{^}

STM{cond}<mode>　Rn{!}，{reglist}{^}

- ◆ cond 是指令的执行条件。
- ◆ mode，总共用 8 种，对初学者而言，只需要掌握 IA、FD 模式即可，其中 IA 表示每次传送后地址加 4，FD 表示满递减堆栈，读者可以结合下面的例子理解关

于满递减堆栈 FD 的使用，在 3.1.2 节“ARM 寻址方式”中的堆栈寻址部分给出详细讲解。

- Rn 为基址寄存器，注意 Rn 不允许为 R15（程序计数器 PC）。
- 后缀“!”表示将最后的地址回写到 Rn 中。
- Reglist 是寄存器列表，可以包含多个寄存器，寄存器按由小到大的顺序排列，当寄存器标号连续时，用“-”连接，如{R0-R7}，当寄存器标号不连续时，用逗号隔开，如{R1，R4，R6}。
- 后缀“^”的用法，读者可以暂时忽略，在第 7 章中分析启动代码时会详细讨论。

例 1：LDMIA R0，{R1-R4}，即将 R0 指向的存储单元（实际上是 4 个字节，因为 ARM 指令是字对齐的）的内容[1]加载到寄存器 R1～R4 中。

如图 3-1 所示，指令的执行过程：首先将 R0 指向的内存单元的数据 51 加载到 R1 寄存器中，然后地址自动加 4（因为传送的数据是 32 位的，因此为 4 个字节）；将 52 加载到 R2 寄存器中，然后地址再自动加 4；将 53 加载到 R3 寄存器中，然后地址再自动加 4；最后将 54 加载到 R4 寄存器中。

注意：*在指令执行过程中，R0 的值并没有变化。此外，由指令的执行过程可以很容易理解 IA 即 Increase After 的意思，通俗一点理解就是：先传送数据，然后更新地址值（Increase the address after transforming the data to the register）。*

例 2：LDMIA R0!，{R1-R4}，即将 R0 指向的存储单元（实际上是 4 个字节，因为 ARM 指令是字对齐的）的内容加载到寄存器 R1～R4 中。

如图 3-2 所示，指令的执行过程：首先将 R0 指向的内存单元的数据 51 加载到 R1 寄存器中，然后地址自动加 4（因为传送的数据是 32 位的，因此为 4 个字节）；将 52 加载到 R2 寄存器中，然后地址再自动加 4；将 53 加载到 R3 寄存器中，然后地址再自动加 4；将 54 加载到 R4 寄存器中，然后地址值再自动加 4，并将地址值赋值给 R0。

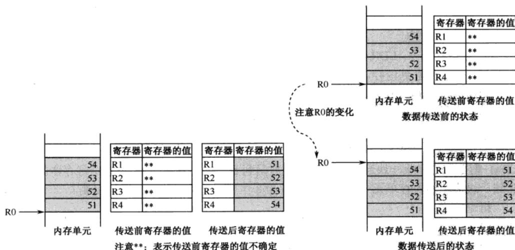


图 3-1 LDMIA 多寄存器传送指令详解　　图 3-2 LDMIA 多寄存器传送指令（R0 值更新）

[1] 此处存储单元中的内容是以小端方式存储的，而且是 4 字节对齐的，但是为了描述指令的执行过程，暂时忽略了这些细节，请读者注意。

2. 数据处理指令

数据处理指令包括数据传送指令（MOV）、算术逻辑运算指令（ADD、SUB、BIC、ORR）和比较指令（CMP、TST）等。

- MOV

寄存器与寄存器之间的数据传送指令，也可以将一个立即数传送给目标寄存器。

例 1：MOV R0，#8，即 R0=8，注意立即数前面需要加#号。当然用 MOV 指令传送立即数时，对立即数是有一定要求的，本书没有详细展开讲解，感兴趣的读者可以查阅相关资料，后面介绍的 LDR 伪指令可用于加载任意的 32 位立即数或地址值到目标寄存器中。

例 2：MOV R0，R1，将 R1 的内容传送到 R0 中，即指令执行完后 R0=R1。

例 3：MOV R0，R1，LSL #3，将 R1 的内容左移三位①，然后传送到 R0，即指令执行完后 R0=（R1<<3）。

例 4：MOV PC，LR，该指令可以实现子程序的返回，其中 PC 和 LR 是 ARM 汇编器对 ARM 的寄存器进行了预先定义，PC 即 R15，LR 即 R14，因此该指令相当于 MOV R15，R14。

- ADD、SUB、BIC、ORR

ADD 是加法指令，SUB 是减法指令，BIC 是位清除指令。位清除指令的基本格式：BIC{cond}{S}Rd，Rn，operand2。指令执行过程是将寄存器 Rn 的值与 operand2 的值的反码按位做逻辑与操作，结果保存到目标寄存器 Rd 中。通俗的理解就是，operand2 中哪些位为 1，则将 Rn 中相应的位清零即可。ORR 是逻辑或运算指令，基本格式：ORR{cond}{S}Rd，Rn，operand2。指令执行过程：将寄存器 Rn 的值与 operand2 的值做逻辑或操作，结果保存到目标寄存器 Rd 中。

例 1：ADD R0，R1，#1，该指令将 R1 的值加 1，然后加载到寄存器 R0 中，即 R0=R1+1。

例 2：ADDS R0，R1，#1，R0=R1+1，注意该指令后面加了个 S，意思是该条指令执行后可能会影响当前程序状态寄存器 CPSR 中的条件标志位。

例 3：BIC R0，R0，#0xF，将 R0 的后 4 位清零，将结果再重新保存到 R0 中。

例 4：ORR R0，R0，#0xF，将 R0 的后 4 位置 1（与“1”相或运算，可以实现置 1 的功能），将结果再重新保存到 R0 中。

- CMP、TST

CMP 是比较指令。指令格式：CMP{cond} Rn，operand2。指令的执行过程：将寄存器 Rn 的值减去 operand2 的值，根据操作的结果更新 CPSR 中相应的条件标志位，以便后面的指令根据相应的条件标志位来判断是否执行。

TST 是位测试指令。指令格式：TST{cond} Rn，operand2。指令的执行过程：将寄存器 Rn 的值与 operand2 的值按位进行逻辑与操作，根据操作的结果更新 CPSR 中相应的条件标志位，以便后面指令根据相应的条件标志位来判断是否执行。

关于这两条指令的使用，请读者参见第 7 章“启动代码分析”。

3. 跳转指令

ARM 跳转指令有 B 和 BL。

① ARM 指令中第二操作数支持移位运算，移位运算有逻辑左移、逻辑右移、算术左移和算术右移等。

- B 跳转指令的基本格式：B{cond} label。基本功能：直接跳转到指定的地址去执行。需要注意的是，使用 B 指令实现程序跳转时，程序的跳转范围为±32Mb。
- BL 是带返回地址的跳转，指令自动将下一条指令的地址复制到链接寄存器 R14(LR) 中，然后跳转到指定的地址去执行，执行完后，返回到跳转前指令的下一条指令处执行。

例：B func1，跳转到 func1 地址处执行。

注意：汇编语言中的标号代表的是地址，即 func1 是代表的地址。

4. 程序状态寄存器访问指令

在 ARM 中，对程序状态寄存器（当前程序状态寄存器 CPSR 和备份程序状态寄存器 SPSR）的操作是通过专门的指令（MSR 和 MRS）来实现的，其他指令不能实现对程序状态寄存器的操作。程序状态寄存器的格式如图 3-3 所示。通过 MRS 与 MSR 配合使用实现对程序状态寄存器的访问，可以通过读—修改—写操作来实现开关中断、切换处理器模式切换等。

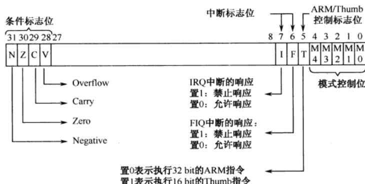


图 3-3 程序状态寄存器

- MRS 读程序状态寄存器指令。指令格式：MRS{cond} Rd，psr。基本功能：将程序状态寄存器 psr 中的内容读入到目标寄存器 Rd 中。
- MSR 写程序状态寄存器指令。指令格式：MSR{cond} psr_fields，#immed_8r 或者 MSR{cond} psr_fields，Rm。

其中，fields 指定了传送的位域。CPSR/SPSR 的位域定义如图 3-4 所示。需要注意的是，ARM 处理器的工作模式分为用户模式和特权模式，除用户模式外的其他 6 种工作模式为特权模式，只有在特权模式下才能对程序状态寄存器（当前程序状态寄存器 CPSR 和备份程序状态寄存器 SPSR）进行修改，在用户模式下不允许修改程序状态寄存器（当在用户模式下试图修改程序状态寄存器时，会产生未定义指令中止异常）。

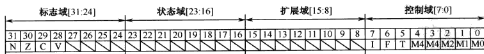


图 3-4 CPSR/SPSR 的位域划分

例 1：将处理器工作模式切换到管理模式。

MSR CPSR_c，#0xD3，将 0xD3 写入到 CPSR 的低 8 位，因为此时 M[4：0]=0b10011（0b 表示数据以二进制的形式表示），所以系统进入管理模式。

例 2：运用“读—修改—写”的方法将处理器的工作模式切换到管理模式。

```
MRS R0，CPSR            ；将 CPSR 的内容读入到 R0
BIC R0，R0，#0x1F       ；将 CPSR 中与模块控制有关的位清零
ORR R0，R0，#0xD3       ；重新修改 CPSR 中模式控制位（其中 ORR 是或运算指令）
MSR CPSR_cxsf，R0       ；将修改后的值回写到 CPSR 中
```

注意：采用“读—修改—写”的方式操作程序状态寄存器是为了防止对 CPSR 中其他位产生影响。

5. 协处理器访问指令

在 ARM 系统中，协处理器 CP15 主要用于存储管理，CP15 总共包含了 16 个 32 位的寄存器，其编号为 C0～C15。在裸机开发中，很少涉及对协处理器的访问（当修改处理器总线模式的时候会用到），在此只做简单的讲解。

访问协处理器的指令为 MCR 和 MRC。

- MRC：协处理器到 ARM 寄存器的数据传送指令。该指令可以将协处理器的寄存器中的数据传送到 ARM 处理器寄存器中。
  指令的基本格式：MRC{cond} p15，0，Rd，CRn，CRm{，opcode2}。
  - ◆ Rd 是 ARM 中的寄存器，作为目标寄存器。
  - ◆ CRn 是协处理器中的寄存器，作为源寄存器，存放第一个操作数其编号为 C0，C1，…，C15。
  - ◆ CRm 是附加的源寄存器，当指令中不需要提供其他信息时，CRm 应指定为 C0。
  - ◆ opcode2 用于提供附加信息，当指令中没有附加信息时，将其制定为 0 即可。
- MCR：ARM 寄存器到协处理器寄存器的数据传送指令。该指令可以将 ARM 处理器寄存器中的数据传送到协处理器的寄存器中。
  指令的基本格式：MCR{cond} p15，0，Rd，CRn，CRm{，opcode2}。
  - ◆ Rd 是 ARM 中的寄存器，作为源寄存器。
  - ◆ CRn 是协处理器中的寄存器，作为目标寄存器，其编号为 C0，C1，…，C15。
  - ◆ CRm 是附加的目标寄存器，当指令中不需要提供其他信息时，CRm 应指定为 C0。
  - ◆ opcode2 用于提供附加信息，当指令中没有附加信息时，将其制定为 0 即可。

例 1：mrc p15，0，r0，c1，c0，0。该指令的功能是将协处理器 c1 中的内容读入到 ARM 处理器 R0 中。

例 2：mcr p15，0，r0，c1，c0，0。该指令的功能是将 ARM 处理器 r0 中的数据写入到协处理器寄存器 c1 中。

提醒读者注意，这里只是简单地介绍了在系统启动代码中用到的与 CP15 协处理器相关的指令，其他的协处理器指令并没有涉及，因为作为入门来说，这两条指令就足够了。

3.1.2 ARM 寻址方式

从 ARM 的寻址方式开始讲起，读者可以尽快地了解 ARM 指令类别，从而有利于快速掌握学习重点。寻址方式即从指令中给出的地址码字段寻找到指令执行所需要的真实操作数的方式。

- 立即寻址

立即寻址指令中的数据就包含在指令中，取出指令的同时也就得到了实际的操作数，即通常所说的立即数。

例：MOV R1，#9 将 9 传送到寄存器 R1 中。

注意：立即数必须以#开头，并且这里的立即数是有一定要求的。

- 寄存器寻址

实际的操作数存放在寄存器中，指令中给出的是寄存器编号，指令执行时直接读取寄存器值。

例：MOV R2，R1。该指令将寄存器 R1 中的数据传送到寄存器 R2 中。假设指令执行前，寄存器 R1 中的值是 0x323（表示十六进制数时要用 0x 前缀），则指令执行后，寄存器 R2 中的值为 0x323。

- 寄存器移位寻址

寄存器移位寻址是 ARM 指令集所特有的寻址方式，当第 2 个操作数是寄存器移位方式时，第 2 个寄存器操作数在与第 1 个操作数结合之前，要先进行移位操作，然后再与第 1 个操作数结合。

例：MOV R2，R1，LSL #3。指令执行时先将寄存器 R1 的值逻辑左移 3 位，然后再将移位后的值传送到寄存器 R2 中。

- 寄存器间接寻址

寄存器间接寻址指令中的地址码给出的是一个通用寄存器编号，而指令中所需要的操作数保存在该寄存器所指向的存储单元中，即寄存器为操作数的地址指针。

例 1：LDR R1，[R0]。假设指令执行前，寄存器 R0 中的值是 0x30000000，则该指令将内存单元 0x30000000 开始的一个字（4 个字节）的内容加载到寄存器 R1 中。注意：默认情况下，ARM 处理器在内存单元中是以小端方式存储数据的，因此，实际上是将 0x30000003～0x30000000 中的内容加载到寄存器 R1 中，如图 3-5 所示，指令执行完后，R1=0xE7FF0010。

例 2：SWP 指令的执行过程分析。SWP 指令用于在内存和寄存器之间字数据交换指令，该指令是一条原子操作（所谓原子操作是指执行过程不可被打断的操作）SWP 指令的执行过程：将一个内存单元（该单元地址放在寄存器 Rn 中）的内容读取到一个寄存器 Rd 中，同时将另一个寄存器 Rm 的内容写入到该内存单元中。

SWP 指令的基本格式：SWP{<cond>}{B} Rd，Rm，[Rn]。

其中，B 后缀可选，若有 B，则表示交换一个字节，否则交换一个字（4 个字节）；Rd 目标寄存器，即将存储器中 Rn 指向的地址单元中的数据加载到该寄存器；Rm 源寄存器，即将该寄存器中的数据存储到存储器中 Rn 指向的地址单元处。SWP 指令的执行过程如图 3-6 所示。

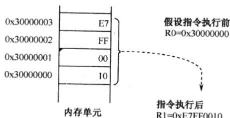


图 3-5 寄存器间接寻址

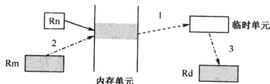


图 3-6 SWP 指令的执行过程

指令执行过程：首先，将 Rn 执行的内存单元中的数据保存到一个临时单元中；然后，将 Rm 寄存器中的数据写入到该内存单元；最后，将临时单元中的数据加载到寄存器 Rd 中。

由上述执行过程可以发现，当 Rm 与 Rn 相同时，该指令可以完成寄存器与存储器间数据的交换。例如：SWP R2，R2，[R1]，该指令可以将 R1 执行的内存单元中的数据和寄存器 R2 中的数据交换。

- 基址寻址

在基址寻址中操作数的实际地址计算方法是：操作数的实际地址=基址寄存器的内容+指令中给出的偏移量。基址寻址常用于访问基址附近的一段存储单元，常用于查表、数组、结构体等数据结构的操作。

例：LDR R1，[R0，#4] 将 R0 中的值加上 4 形成地址，将此地址中的值加载到寄存器 R2 中，基址寻址过程如图 3-7 所示。注意：默认情况下，ARM 是以小端方式存储数据，所以实际上是将 0x30000004～0x30000007 中的数据加载到寄存器 R1 中。

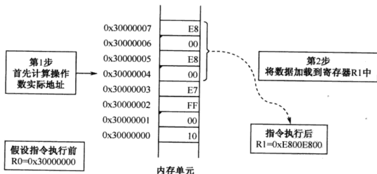


图 3-7 基址寻址过程

- 多寄存器寻址

多寄存器寻址就是一次可以传送几个寄存器值。请读者参见 3.1.1 节“ARM 指令集”中讲解的存储器访问指令内容。

- 堆栈寻址

堆栈是按特定顺序进行访问的存储区，对 ARM 处理器来说是满递减堆栈，即堆栈指针堆栈的栈顶。

例 1：STMFD SP!，{R0-R2}。该指令将寄存器 R0～R2 中的数据压入堆栈。注意：“!”

说明最后堆栈指针更新。假设指令执行前 R0=0x55、R1=0x77、R2=0x33，堆栈指针 SP=0x3000000C。

指令执行过程：首先，堆栈指针 SP 减 4（因为 ARM 指令是 32 位的，一次传送 4 个字节），即此时 SP=0x30000008，将寄存器 R2 中的数据 0x33 入栈；然后，堆栈指针 SP 再减 4，即此时 SP=0x30000004，将寄存器 R1 中的数据 0x77 入栈；最后，堆栈指针 SP 再减 4，即此时 SP=0x30000000，将寄存器 R0 中的数据 0x55 入栈。入栈操作如图 3-8 所示。由上面的分析知，堆栈指针始终指向最后一个入栈的数据（注意，结合 ARM 堆栈是满递减的含义）。

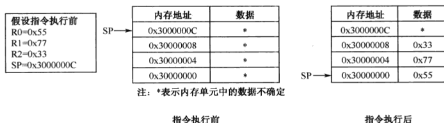


图 3-8　入栈操作

例 2：LDMFD SP!，{R0-R2}。该指令的含义是，数据出栈，放入 R0～R2 寄存器中。注意："！"说明最后堆栈指针更新。假设指令执行前堆栈指针 SP=0x30000000，出栈操作如图 3-9 所示。

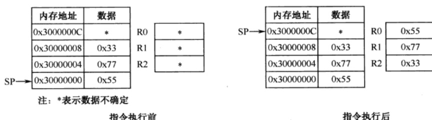


图 3-9　出栈操作

指令执行过程：首先，0x55 出栈保存到寄存器 R0 中，堆栈指针 SP 加 4（因为 ARM 指令是 32 位的，一次传送 4 个字节），此时 SP=0x30000004；然后，0x77 出栈保存到寄存器 R1 中，堆栈指针 SP 再加 4，此时 SP=0x30000008；最后，0x33 出栈保存到寄存器 R0 中，堆栈指针再加 4，此时 SP=0x3000000C。

由上述分析可知，对于入栈和出栈指令而言，在寄存器列表中，寄存器的标号必须按照由小到大的顺序排列。但是请读者注意，入栈时，编号大的寄存器中的数据先入栈。例如，STMFD SP!，{R0-R2}，R2 中的数据先入栈，然后按照编号递减的顺序依次入栈；出栈时，数据出栈，并保存到编号最小的寄存器。例如，LDMFD SP!，{R0-R2}，数据出栈，存入寄存器 R0 中，然后数据依次出栈，按照编号依次递增的顺序保存到相应的寄存器中。因此从这个意义上说，STMFD 相当于 STMDB，LDMFD 相当于 LDMIA（此处后缀 DB 和 IA 指令的执行模式）。

3.1.3 ARM伪操作和伪指令介绍

ARM 汇编程序由指令（ARM 指令和伪指令）、伪操作和宏指令组成。伪指令是汇编程序对源程序进行汇编期间由汇编程序将其替换成合适的 ARM 指令或 Thumb 指令。宏是一段独立的程序代码，类似于 C 语言中用 define 定义的宏，它是通过伪指令定义的。当程序被汇编时，汇编程序将对每个调用进行展开，用宏定义取代源程序中的宏指令。

1. ARM 伪操作

ARM 伪操作主要包括符合定义伪操作、数据定义伪操作、汇编控制伪操作、信息报告伪操作等。本节只介绍部分伪操作，主要用在启动代码的编写。对于其他伪操作，读者可以查阅相关技术手册。

- GET

GET 通常用于包含定义常量的源文件，如用 EQU 定义的外设地址，类似于 C 语言中用 include 包含头文件。

例：GET 2440addr.inc 将 2440addr.inc 文件包含到文件中。

注意：汇编语言中被包含的文件常以.inc 结尾。

- AREA、ENTRY 和 END

一个 ARM 汇编程序可分为几个段，如数据段、代码段、堆栈段等，AREA 操作常用于定义一个段。通常，一个 ARM 源程序至少需要一个代码段，大的程序可以包含多个代码段及数据段。汇编程序分段设计，有利于实现分散加载机制。如果读者对分散加载机制不是很了解，建议在初学阶段不必深究。ENTRY 用于指定程序的入口点。END 用于告诉汇编编译器源文件已经结束。AREA、ENTRY 和 END 的用法如图 3-10 所示。

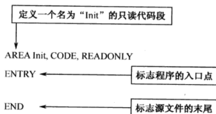


图 3-10 AREA、ENTRY 和 END 的用法

- EXPORT 和 IMPORT

EXPORT 伪操作用于声明外部标号，即当前标号是本源文件中定义的，在其他文件中可能会被引用。

IMPORT 伪操作用于告诉编译器当前的符号不是在本源文件中定义的，而是在其他源文件中定义的，在本源文件中可能引用该符号。

例 1：IMPORT |Image$$RO$$Base| 告诉编译器|Image$$RO$$Base|是在其他文件中定义的，本文件可能引用该符号。实际上，|Image$$RO$$Base|是编译器产生的一个符号，用于表示 RO 段的结束地址。

此外，在 ARM 汇编源文件中可以用 EXPORT 伪操作声明一个外部标号，即当前标号是本源文件中定义的，在其他文件中可能会被引用。

例 2：IMPORT RdNF2SDRAM 告诉编译器 RdNF2SDRAM 是在其他文件中定义的，本文件可能引用该标号。

例 3：EXPORT StartPointAfterSleepWakeUp 告诉编译器 StartPointAfterSleepWakeUp 是本文件中定义的，其他文件可能引用该标号。

● EQU

用于定义常量。例如，可以用来定义外设的地址。

提醒读者注意，在每条 ARM 指令前必须有空格，但是用 EQU 定义常量时，必须顶格写，否则编译器报错。如图 3-11 所示，第一行 USERMODE EQU 0x10 没有顶格写，编译器报错：Unknown opcode。

```
 USERMODE      EQU     0x10
FIQMODE        EQU     0x11
IRQMODE        EQU     0x12
```

```
Error   : A1163E: Unknown opcode
2440init.s line 15
```

图 3-11　EQU 定义常量编译器报错情况

● LTORG

LTORG 用于声明一个文字池，所谓的文字池就是一个数据缓存区。在使用 LDR 伪指令时，要在适当的地址加入 LTORG 声明文字池，这样就会把要加载的数据保存在文字池内，当用到该数据时会从文字池里面取出相应的数据。文字池的使用请参见后面有关 LDR 伪指令的内容。

● ALIGN

ALIGN 伪操作通过调整地址指针使当前位置满足一定的对齐方式。在 ARM 代码中要求地址标号是字对齐的。

● MACRO 和 MEND

MACRO 和 MEND 伪操作用于宏定义。MACRO 表示宏定义的开始，MEND 表示宏定义的结束。用 MACRO 及 MEND 定义的一段代码，称为宏体。这样，在程序中就可以通过宏指令多次调用该代码段。

宏定义的基本格式如下：

```
MACRO
{$label} MacroName {$parameter} {$parameter}…
; 这里添加自己的代码
MEND
```

◆ $label 代表一个标号，在宏展开时替换成相应的值。

◆ MacroName 用于指定宏的名称。

◆ $parameter 代表需要传递的参数，类似于 C 语言中的形式参数。

其中，{}中的项表示是可选的。

例：有下面一个宏定义（读者可以不必关心该宏实现的功能，在第 7 章中会详细讲解，在此只需要关注从宏定义到宏展开是如何实现的即可）：

```
MACRO
$Label HANDLER $HandleAddr
$Label
    sub     sp,sp,#4
    stmfd   sp!,{r0}
    ldr     r0,=$ HandleAddr
    ldr     r0,[r0]
```

```
        str     r0,[sp,#4]
        ldmfd   sp!,{r0,pc}
    MEND
```

对比上面的宏定义可知：该宏的名字是 HANDLER，调用该宏时需要一个参数$HandleAddr。

在程序中可以通过如下方式调用该宏：HandlerIRQ　HANDLER　HandleIRQ。

宏展开的结果如下：

```
HandlerIRQ
        sub     sp, sp, #4
        stmfd   sp!, {r0}
        ldr     r0, = HandleIRQ
        ldr     r0, [r0]
        str     r0, [sp, #4]
        ldmfd   sp!, {r0, pc}
```

由宏展开的结果可以看出，HandlerIRQ 代替了宏定义中的标号$HandlerLabel，同时传递给宏的参数 HandleIRQ 代替了宏定义中的形式参数$ HandleAddr。

- MAP 和 FIELD

MAP 用于定义内存表的首地址，其中 MAP 可以用^表示。FIELD 用于定义一个内存表中的数据域，其中 FIELD 可以用#表示。MAP 和 FIELD 组合使用类似于 C 语言中定义一个数组，MAP 指向数组的首地址，FIELD 用于分配数组中的各个元素。读者可以结合下面的例子加以理解。

例：由如下一个内存表定义：

```
1   _ISR_STARTADDRESS   EQU   0x33FFFF00
2   MAP    _ISR_STARTADDRESS
3   HandleReset         FIELD     4
4   HandleUndef         FIELD     4
5   HandleSWI           FIELD     4
6   HandlePabort        FIELD     4
7   HandleDabort        FIELD     4
```

第 1 行，用 EQU 定义了一个常量_ISR_STARTADDRESS。

第 2 行，用 MAP 指定了该内存表的首地址是_ISR_STARTADDRESS，即内存表的首地址为 0x33FFFF00。

第 3 行，用 FIELD 指定了内存表的第 1 个元素是 HandleReset，后面的 4 表示该元素的地址为从内存表的首地址开始占据 4 个字节的长度，即 HandleReset 地址范围为 0x33FFFF00--0x33FFFF03。

第 4 行，用 FIELD 指定了内存表的第 2 个元素是 HandleUndef，后面的 4 表示该元素占据 4 个字节的长度。即 HandleReset 地址范围为 0x33FFFF03--0x33FFFF07。

其他依此类推，最后，在内存中分配的内存表如图 3-12 所示。



图 3-12　内存表

此外，由于 MAP 和^是等价的，FIELD 也可以用#代替，因此也可以通过如下的方式定义上述内存表。通过这两种方式定义的内存表的含义完全一样。

```
1    _ISR_STARTADDRESS   EQU   0x33FFFF00
2    ^    _ISR_STARTADDRESS
3    HandleReset          #     4
4    HandleUndef          #     4
5    HandleSWI            #     4
6    HandlePabort         #     4
7    HandleDabort         #     4
```

引申：建立上述内存表后，可以从 C 源文件中通过如下方式访问。

```
#define pISR_SWI        (*(unsigned *)(_ISR_STARTADDRESS+0x8))
```

首先通过强制类型转换将(_ISR_STARTADDRESS+0x8)转换成指向 unsigned 型数据的指针，然后通过指针运算符*（通常，*也称为间接访问运算符）取其指向的内容。在 C 源文件中对 pISR_SWI 进行赋值（通常将一个函数地址赋给它）时，就将函数的地址填到了存储单元 0x33FFFF08 处，这样在汇编语言文件中访问地址 0x33FFFF08 时，就可以实现对 C 源文件中函数的调用（读者可以结合后面的具体实验进行理解）。

2. ARM 伪指令

读者应首先搞清楚什么是伪指令。对于 ARM 指令或者 Thumb 指令，经过编译器编译后会生成相应的机器指令，然后在运行时就可以执行。但是，伪指令是汇编程序对源程序进行汇编处理期间由汇编程序处理，在汇编时会被适当的 ARM 指令或 Thumb 指令代替。

常用的 ARM 伪指令有中等范围地址读取伪指令 ADRL 以及大范围的地址读取伪指令 LDR。

- ADRL

ADRL 伪指令的基本格式：ADRL{cond} register，expr。ADRL 伪指令通常被编译器替换成两条合适的指令，来实现其功能。读者可以结合下面的例子加以理解。

```
    AREA    Init,CODE,READONLY
    ENTRY
    ADRL R1,varible1

varible1 DCD 5
varible2 DCD 6

    END
```

图 3-13　ADRL 伪指令

例：按照第 2 章讲解的步骤，先建立一个工程，然后建立一个汇编语言源文件 test.s，最后将 test.s 添加到工程中。其中，在 test1.s 中输入如图 3-13 所示的内容。提醒读者注意，DCD 是 ARM 汇编语言中用于定义数据的伪操作，当用 DCD 定义数据时需要顶格写，否则编译器会报错。

然后在 test.s 上单击鼠标右键，选择“Disassemble”（如图 3-14 所示），此时会自动弹出 Disassembly test.s 窗口，如图 3-15 所示。

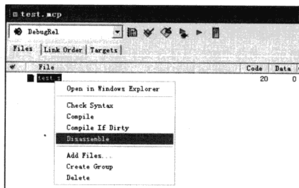
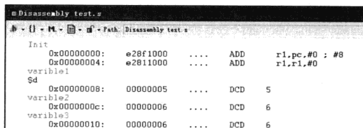


图 3-14 选择“Disassemble”



```
Init
    0x00000000:    e28f1000    ....    ADD    r1,pc,#0 ; #8
    0x00000004:    e2811000    ....    ADD    r1,r1,#0
varible1
$d
    0x00000008:    00000005    ....    DCD    5
varible2
    0x0000000c:    00000006    ....    DCD    6
varible3
    0x00000010:    00000006    ....    DCD    6
```

图 3-15 Disassembly test.s（反汇编）窗口

对比 test.s 文件可以发现，ADRL R1，varible1 被编译器替换成两条指令，分别是 ADD r1，pc，#0 和 ADD r1，r1，#0。这是什么意思呢？

虽然 ARM9 是 5 级流水线，但是指令执行阶段处于流水线的第三级（读者可以不必细究，暂且记住就可以），所以当前正在执行的指令的地址等于 PC 值减去 8。如图 3-15 所示，第一列表示指令的地址，可见第一条指令 ADD r1，pc，#0 的地址是 0x00000000，因此此时 PC 值为 0x00000008，即当前指令下两条指令的地址。ADD r1，pc，#0 执行完后，寄存器 R1=0x00000008。下一条指令 ADD r1，r1，#0 的功能是将 R1 的值加 0 再赋给 R1，执行完这两条指令后，寄存器 R1 中的值是 0x00000008，而这恰好是变量 varible1 的地址（见图 3-15，这也说明了 ARM 汇编语言中标号代表的是一个地址）。

对于上面的分析，读者可能有疑问：ADD r1，r1，#0 这一句不是多余的吗？只能说这是由 ARM 汇编语言编译器决定的，在处理 ADRL 指令时，ARM 汇编语言编译器总是将其替换成两条指令，即使替换成一条指令也可以完成相关的功能，但是编译器还是将其替换成两条指令。

此外，ARM 汇编语言编译器将 ADRL 伪指令替换成相应的指令时，是基于当前的 PC 值进行调整，这有利于产生位置无关代码[①]，在本书第 9.2 节的 SDRAM 实验过程中，读者会再次注意到此功能。

引申：细心的读者可能注意到在上述内容中只介绍了图 3-15 中的第一列，那么第二列

① 有关位置无关代码，读者可以自行查阅有关资料学习，初学者可以暂时忽略这一内容。

表示什么意思呢？e28f1000 又是什么意思呢？e28f1000 和 ADD r1，pc，#0 有什么关系呢？其实，这涉及 ARM 指令集中从 ARM 指令到对应的机器码之间的转换问题。ARM 指令的编码格式如图 3-16 所示。对各个位的具体含义不做统一的概述，只是针对 ADD r1，pc，#0 这一条指令进行具体分析。相信读者看过这一条指令的分析后，对照着指令编码手册，分析其他指令将不再是一件很困难的事情。

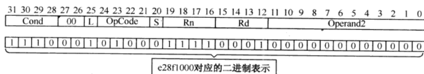


图 3-16　ARM 指令的编码格式

- Cond[31:28]表示指令执行的条件，1110 表示指令无条件执行。
- L[25]当机器码的低 12 位是立即数时，该位为 1；当机器码的低 12 位是寄存器时，该位为 1。
- S[20]表示指令执行过程中是否影响 CPSR 中相应的位，此处 S 为 0，表示该指令的执行不影响 CPSR 中相应的位。
- OpCode[24:21]表示指令助记符对应的操作码，0100 表示是 ADD 指令。
- Rn[19:16]表示源寄存器号，此处为 1111，即寄存器 R15，也就是程序计数器 PC。
- Rd[15:12]表示目标寄存器号，此处是 0001，即寄存器 R1。
- Operand2[11:0] 表示指令的第二操作数，此处是 0。

因此，机器码 e28f1000 中各位的含义如图 3-17 所示，即 e28f1000 对应的指令为 ADD r1，pc，#0。同理，可知图 3-15 中机器码 e2811000 对应的指令为 ADD r1，r1，#0。

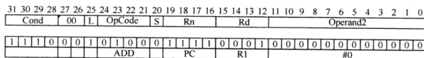


图 3-17　机器码 e28f1000 中各位的含义

- LDR

LDR 伪指令实现将一个 32 位常数或者地址值加载到寄存器中。

LDR 伪指令基本格式如下：

```
LDR{cond} register, =[expr|label]
```

其中，cond 表示指令的执行条件；expr 为 32 位的常量，label 表示地址表达式或者外部表达式。下面通过一个具体例子介绍 LDR 伪指令的具体用法。

例 1：还是采用上面的例子，只是将 ADRL R1，varible1 这一句改为 LDR R1，=varible1，test.s 中的内容如图 3-18 所示。

然后在 test.s 上单击鼠标右键，选择“Disassemble”（如图 3-19 所示），此时会自动弹出 Disassembly test.s 窗口，如图 3-20 所示。

```
        AREA    Init,CODE,READONLY
        ENTRY
        LDR R1,=varible1
varible1 DCD 5
varible2 DCD 6
varible3 DCD 6
        END
```

图 3-18　LDR 伪指令

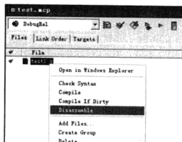


图 3-19　选择“Disassemble”

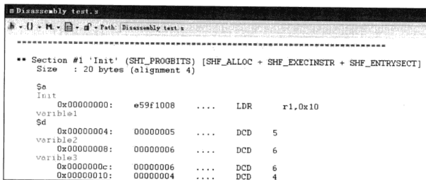


图 3-20　Disassembly test.s 窗口

由图 3-20 可以看到，ARM 汇编语言编译器将 LDR R1，=varible 伪指令替换成了 LDR r1,0x10 指令。提醒读者注意，这里虽然都用了 LDR 指令，但是前一个是 ARM 伪指令，后面一个才是寄存器加载指令 LDR。两者的区别是，当 LDR 是伪指令时，后面加载的常量或者地址标号前面必须有一个“=”。LDR r1,0x10 表示将地址 0x00000010 中的数据加载到寄存器 r1 中。但是，0x00000010 是在哪里定义的呢？从图 3-20 中可以看到，地址 0x00000010 处放置了一条数据定义伪操作 DCD　4，也就是说，地址 0x00000010 中的数据是 4。这个 4 又是哪里来的呢？细心的读者可能已经发现，变量 varible1 的地址就是 0x00000004。结合前面对文字池的描述，可以得出下面的结论：用 LDR 伪指令向寄存器中加载数据时，在汇编期间，编译器用一条合适的指令（如 LDR r1,0x10）代替 LDR 伪指令，同时在文件的末尾分配一个文字池，即上面的 0x00000010 就是编译器自动分配的一个文字池（也就是一个数据缓冲区），然后将 varible1 的地址 0x00000004 暂时存放到这个文字池中，当执行到指令 LDR r1,0x10 时，该指令会从文字池中取出 varible 的地址并将其加载到寄存器中。

提醒读者注意：虽然编译器会自动分配文字池，但是 LDR 伪指令要求文字池与 LDR 伪指令的距离在前后 4KB 范围内，因此当文字池距离 LDR 伪指令较远时应自定义一个文字池，在汇编语言中用关键字 LTORG 即可分配一个文字池。

例 2：自定义文字池。按照第 2 章讲解的步骤，先建立一个工程，然后建立一个汇编语言源文件 test.s，最后将 test.s 添加到工程中。其中在 test.s 中输入如图 3-21 所示的内容。

然后在 test.s 上单击鼠标右键，选择“Disassemble”（如图 3-22 所示），此时会自动弹出 Disassembly test.s 窗口，如图 3-23 所示。

```
        AREA      Init,CODE,READONLY
        ENTRY
        LDR R1,=varible1
        LTORG
varible1 DCD 5
varible2 DCD 6
varible3 DCD 6

        END
```

图 3-21　test.s 的内容

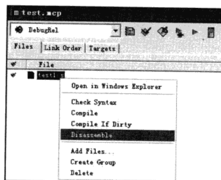


图 3-22　选择“Disassemble”

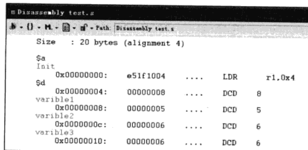


图 3-23　Disassembly test.s 窗口

对比图 3-20 可以看到，用 LTORG 自定义一个文字池，该文字池的地址是 0x00000004，那么伪指令 LDR R1，=varible 被相应地替换成 LDR r1,0x4。此时，文字池中存放的数据是 8，即变量 varible 的地址 0x00000008，因此通过指令 LDR r1,0x4 就可以将文字池中的数据取出来加载到寄存器 R1 中，即将变量 varible 的地址加载到寄存器 R1 中。

3.2　ARM 汇编基础知识

ARM 汇编语言的基础知识主要涉及数据定义、循环控制等。数据定义主要是描述在内存中为特定数据分配相应内存单元的方式，可以分配一个字节，也可以分配一个字，这其中设计较多的伪操作有 DCD、DCB、DCW 等。但是对于启动代码阶段，只需要掌握 DCD 和 SPACE 的用法即可，其他的数据定义方式无非是分配内存单元的长度不同或者是可以选择在分配内存单元是否进行初始化等，对此，初学者可以暂时忽略，将重点放在对 DCD 和 SPACE 的理解上。掌握这两种伪操作之后，其他的伪操作都是一样的道理。汇编语言中的循环控制与高级语言中的循环控制都是一样的道理，在启动代码阶段没有涉及，因此，在此并没有展开讨论。

ARM 汇编程序中数据定义主要用到以下几个伪操作。

- DCD

DCD 伪操作的基本格式如下：

{label} DCD expr{,expr}{,expr}…

DCD 常用于分配一块连续的内存单元，并用 expr 初始化。其中需要注意的是，label 代表所分配的内存单元的地址。在上面的例子中，varible1、varible2 和 varible3 都是表示内存单元的地址，如图 3-24 所示，其中 varible1 为 0x00008008，varible1 为 0x0000800C，varible1 为 0x00008010。

```
00008000 [0xe51f1004]   ldr    r1,0x00008004 ;
00008004 [0x00008008]   dcd    0x00008008   ...
varible1 [0x00000005]   dcd    0x00000005   ...
varible2 [0x00000006]   dcd    0x00000006   ...
varible3 [0x00000006]   dcd    0x00000006   ...
00008014 [0xe800e800]   dcd    0xe800e800   ...
```

图 3-24　变量名代表变量的地址

- SPACE

SPACE 用于分配一块内存单元，并将其初始化为 0。SPACE 伪指令的基本格式为：{label} SPACE expr，label 表示内存块的起始地址，expr 表示所要分配的内存字节数目。

例：在内存中分配 12 个字节的存储单元并初始化为 0。

zero SPACE 12

如图 3-25 所示，分配了 12 个字节长度的连续的内存单元，并用零初始化该内存块。同时，zero 的值是 0x00000014，即该内存块的首地址。

```
Disassembly test.s
Path: Disassembly test.s
    0x00000010:    00000006    ...    DCD    6
zero
    0x00000014:    00000000    ...    DCD    0
    0x00000018:    00000000    ...    DCD    0
    0x0000001c:    00000000    ...    DCD    0
```

图 3-25　SPACE 的用法

3.3　ARM 汇编程序的基本结构

在前面的例子中或多或少地涉及了部分汇编程序，在这一节中将对 ARM 汇编程序进行较细致的讲解。ARM 汇编语言源文件是由不同的段（Section）组成的，常见的有代码段、数据段等。代码段主要存放执行的代码，数据段存放代码执行过程中需要的数据。

在用 ADS 1.2 进行基于 ARM 处理器的程序开发过程中，ARM 源程序文件主要有以下几种类型。

- *.s 表示该文件是一个汇编语言源文件。
- *.inc 表示该文件是一个被汇编语言源文件包含的文件。
- *.c 表示该文件是一个 C 语言源文件。
- *.h 表示该文件是一个头文件。

3.3.1 编写汇编程序基本的格式规范

在编写 ARM 汇编语言源程序时要遵循一定的规范，否则编译器会报错。

- 在 ARM 汇编程序中，所有标号必须在一行的顶格书写。
- 在 ARM 汇编程序中，所有的指令均不能顶格书写，指令前面应该有空格，一般用 Tab 键缩进。
- 因为 ARM 汇编器对标志符大小写是敏感的，因此书写标号及指令时，字母大小写要一致。在 ARM 汇编程序中，指令、寄存器名可以全部为大写字母，也可以全部为小写字母，但不要大小写混合使用。
- 在 ARM 汇编程序中运行使用注释，注释内容由“;”开始一直到此行结束，注释可以顶格书写。
- 为了增加源程序的可读性，在完成不同功能的代码段之间可以适当地插入空行。
- 当单行指令太长时，可以使用字符“\”实现分行，“\”后不能有任何字符。
- 定义变量、常量时，其标志符必须在一行的顶格书写，否则编译器报错。

例 1：标号没有顶格书写，编译器会报错。

如图 3-26 所示，标号 START 没有顶格书写，在编译时，编译器报错。

注意：标号 START 左边的红色小箭头指示哪一行出错。

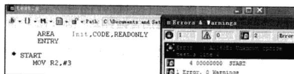


图 3-26　标号未顶格书写，编译器报错

例 2：指令顶格书写，编译器会报错。

如图 3-27 所示，指令 MOV R2，#3 顶格书写，在编译时，编译器报错误。左边的红色小箭头指示了出错的行，同时可以注意到，标号 START 已经顶格书写了，所以编译器报错。

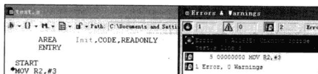


图 3-27　指令顶格书写，编译器报错

例 3：指令大小写混写，编译器会报错。

如图 3-28 所示，指令 Mov R2，#3 中 Mov 指令大小写混写，在编译时，编译器报错，左边的红色小箭头指示了出错的行。

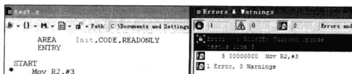


图 3-28　指令大小写混写，编译器报错

例 4：在汇编语言源程序中使用注释。

如图 3-29 所示，注释以分号开始，提醒读者注意：输入分号时应在半角状态下输入，否则编译器报错。

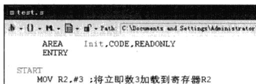


图 3-29　注释的使用

例 5：定义变量时，变量的标号要顶格书写，否则编译器报错，如图 3-30 所示。

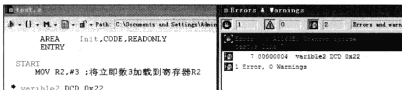


图 3-30　变量标号未顶格书写，编译器报错

3.3.2　程序入口和程序结束

程序的入口点用 ENTRY 伪操作指定。伪操作的格式：ENTRY。END 伪操作用于告诉汇编编译器源文件已结束。每一个汇编源文件均要使用一个 END 伪操作，指示本源程序结束，如图 3-31 所示。

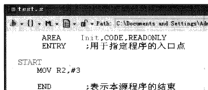


图 3-31　ARM 汇编程序基本结构

3.3.3　段

ARM 汇编语言源文件是由不同的段（Section）组成的，常见的有代码段、数据段等，代码段主要存放执行的代码，数据段存放代码执行过程中需要的数据。通常，一个 ARM 源程序至少需要一个代码段，大的程序可以含多个代码段及数据段①。每个段可以用 ARM 伪操作 AREA 定义，同时还可以定义各个段的属性，常见的段属性有 READONLY（只读）、READWRITE（可读/写）。当用 AREA 定义一个段时，如果段名以数字开头，则段名需要

① 一个程序包含多个代码段数据段的情况主要用在程序分散加载的情况下，初学者可以暂时忽略。在本书后面的实验部分，整个程序只是定义了一个代码段，这对于初学者来说就足够了。如果读者想了解更多的情况，请参见 ADS 链接手册和 ARM 程序映像文件的组成结构。

用“|”括起来，如|1_tst|。

例：用 AREA 定义段（如图 3-32 所示）。

由图 3-32 可知，该汇编语言程序有两个段组成，即 Init 段和 Data 段。

```
1      AREA     Init,CODE,READONLY
2      ENTRY
3
4 START
5     MOV R2,#3
6
7     AREA Data,DATA,READWRITE
8 varible DCD 12
9     END
```

图 3-32　AREA 使用示例

第 1 行，用 AREA 伪操作定义了一个 Init 段，后面的 CODE 表示该段是一个代码段，READONLY 表示该段的属性为只读。

第 2 行，用 ENTRY 指定程序的入口点。

第 4 行，定义了一个标号 START，并且顶格书写。

第 5 行，书写了一条指令，注意：指令并没有顶格书写。

第 7 行，用 AREA 伪操作定义了一个 Data 段，后面的 DATA 表示该段是一个数据段，READWRITE 表示该段的属性为可读/写。

第 3 行和第 6 行，插入了空行，目的是为了增强程序的可读性。

3.3.4 标号（标志符）

在 ARM 汇编中，标号代表一个地址，段内标号的地址是相对于本段段基址的一个偏移，是在汇编时确定的，而段外标号的地址是在程序连接时确定的。常见的标号主要有基于程序计数器 PC 的标号、绝对地址和局部标号。

- 基于程序计数器 PC 的标号

基于程序计数器 PC 的标号是指位于指令前的标号或者程序中的数据定义伪操作前的标号，这种标号在汇编时被处理成将程序计数器 PC 值加上或减去一个数字常量。它常用于表示跳转指令的目标地址，或者表示代码段中所使用的少量数据。

例 1：按照第 2 章讲解的步骤，先建立一个工程，然后建立一个汇编语言源文件 test.s，最后将 test.s 添加到工程中。其中在 test.s 中输入如图 3-33 所示的内容。

然后在 test.s 上单击鼠标右键，选择“Disassemble”项（如图 3-34 所示），此时会自动弹出 Disassembly test.s 窗口，如图 3-35 所示。

```
     AREA     Init,CODE,READONLY
     ENTRY

START
     LDR R2,=varible
LOOP
     MOV R1,#2
     B LOOP


varible DCD 5
     END
```

图 3-33　编辑 test.s 文件

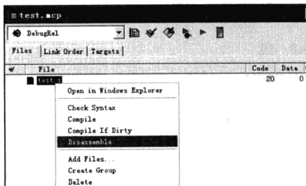


图 3-34　选择“Disassemble”项

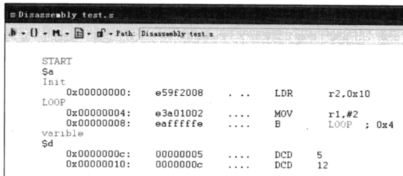


图 3-35 Disassembly test.s 窗口

基于程序计数器 PC 的标号是指位于指令前的标号或者程序中的数据定义伪操作前的标号。因此，在 test.s 中位于指令前的标号有 START 和 LOOP（均顶格书写），位于数据定义伪操作 DCD 前的标号是 varible。这种标号在汇编时被处理成将程序计数器 PC 值加上或减去一个数字常量。但是，从图 3-35 中可以看到，LDR R2，=varible 被处理成 LDR r2,0x10，并没有被处理成将程序计数器 PC 值加上或减去一个数字常量，为什么呢？

从图 3-35 可以发现，在地址 0x00000010 处定义了一个 4 字节的存储单元（DCD 伪操作分配数据是以 4 个字节为单位分配的），并且该存储单元的数据是 12，那么 12 是哪里来的呢？前面曾经讲过用 LDR 伪指令将数据加载到寄存器中时，是通过文字池来实现的，这个 0x00000010 就是编译器分配的文字池，里面的数据 12 就是变量 varible 的地址 0x0000000C。因此，图 3-35 中的 LDR r2,0x10，就是将 0x10 单元的数据 12 加载到寄存器 R2 中，而 12（也就是 0x0000000c）正好是 varible 所代表的地址。

此外，还可以用 AXD 调试器[①]观察反汇编的情况。AXD 调试器主界面如图 3-36 所示。

由前面的分析可知，在 test.s 中定义了三个标号：位于指令前的标号有 START 和 LOOP（均顶格书写），位于数据定义伪操作 DCD 前的标号是 varible。这三个标号代表的是一个地址，在图 3-36 中，在左边汇编窗口中可以很容易地计算出 START 代表的地址是 0x00008000（0x00007ffc+4），LOOP 代表的地址是 0x00008004，varible 代表的地址是 0x8000000c。

到此可以清楚地看到，在汇编语言中，标号确实代表了一个地址值，LDR R2，=varible 并没有被处理成将程序计数器 PC 值加上或减去一个数字常量。这又是为什么呢？请看下面的例子。

例 2：按照第 2 章讲解的步骤，先建立一个工程，然后建立一个汇编语言源文件 test.s，最后将 test.s 添加到工程中。其中，在 test.s 中输入如图 3-37 所示的内容，对比例 1 可以发现，只是将 LDR R2，=varible 替换成 ADRL R2，varible。

然后在 test.s 上右键单击，选择“Disassemble”（如图 3-38 所示），此时会自动弹出 Disassembly test.s 窗口，如图 3-39 所示。

① 关于 AXD 调试器的使用，请参见本章用 AXD 在线仿真调试 ARM 汇编指令实验部分。

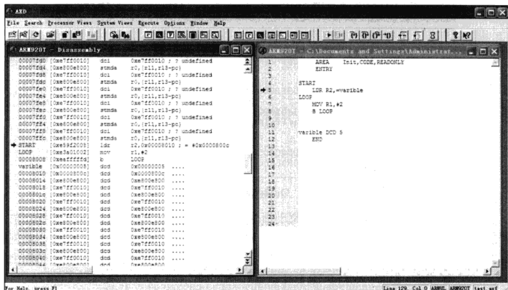


图 3-36　AXD 调试器主界面

```
        AREA     Init,CODE,READONLY
        ENTRY

START
        ADRL  R2,varible
LOOP
        MOV R1,#2
        B LOOP


varible DCD 5
        END
```

图 3-37　test.s 文件内容

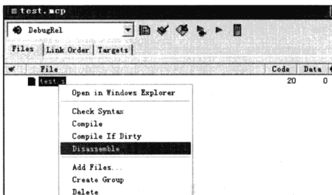


图 3-38　选择“Disassemble”

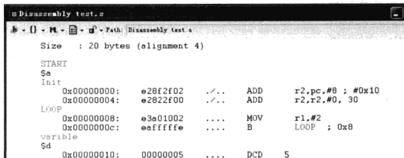


图 3-39　Disassembly test.s 窗口

由图 3-39 可以看到，伪指令 ADRL R2，varible 被替换成两条 ADD 指令，当执行指令 ADD r2,pc,#8 时，由于 ARM 指令流水线的影响，程序计数器 PC 值实际上是当前指令地址加 8，即 0x00000008；然后执行完指令 ADD r2,pc,#8 后，寄存器 r2 的值是 0x00000010（0x00000008+8），而 0x00000010 恰好就是 varible 代表的地址，下一条指令 ADD r2,r2,#0 实际上什么都没做（这是由 ARM 汇编语言编译器决定的，在处理 ADRL 指令时，ARM 汇编语言编译器总是将其替换成两条指令，即使替换成一条指令也可以完成相关的功能，但是编译器还是将其替换成两条指令）。

到此为止，读者可能已经发现标号 varible 确实被处理成将程序计数器 PC 值加上或减去一个数字常量。

通过上面两个例子，可以得出如下结论：在处理 LDR 伪指令时，是借助文字池来实现将数据加载到寄存器中的；在处理 ADRL 伪指令时，标号才被处理成将程序计数器 PC 值加上或减去一个数字常量。这就是 ADRL 可以产生位置无关代码而 LDR 伪指令却不能产生位置无关代码的真正原因。

● 绝对地址

绝对地址是一个 32 位的数字量，它可以用来寻址整个内存空间。当向程序计数器 PC 直接赋值时，可以实现程序的跳转。

例：LDR PC，=0x30000000 对 TQ2440 开发板而言，内存的地址是 0x30000000，因此这条语句的功能是，程序跳转到内存中去执行。

● 局部标号

局部标号主要用于局部范围代码中。局部标号是一个 0～99 的十进制数字，可重复定义。局部变量的作用范围为当前段。

局部标号引用格式：%{F|B}　N（0～99 的十进制数字）。

其中：

◆ %表示引用程序中定义的局部标号。

◆ F 指示编译器只向前搜索被引用的局部标号（其实就是 Forward）。

◆ B 指示编译器只向后搜索被引用的局部标号（其实就是 Backword）。

```
1  InitRam
2    ldr r2, BaseOfBSS
3    ldr r3, BaseOfZero
4  0
5    cmp r2, r3
6    ldrcc r1, [r0], #4
7    strcc r1, [r2], #4
8    bcc %B0
9
10   mov r0, #0
11   ldr r3, EndOfBSS
12 1
13   cmp r2, r3
14   strcc r0, [r2], #4
15   bcc %B1
```

图 3-40　局部标号应用实例

例：局部标号应用实例（如图 3-40 所示的一个汇编语言程序段）。

第 4 行，定义了一个局部标号 0。

第 8 行，对于指令 bcc，本身这是一条跳转指令 b，后面的 cc 指明了跳转指令执行的条件由 3.1.1 节的介绍可知，cc 表示无符号数小于，即第 5 行比较寄存器 r2 和寄存器 r3 中的数据大小，当寄存器 r2 中的数小于寄存器 r3 中的数时，跳转指令执行。跳转到哪里执行呢？%B0 指明了跳转指令的地址，其中%表示对局部标号 0 的引用，B 指示编译器向后搜索局部标号 0，因此指令跳转到第 5 行开始执行。

第 12 行，定了一个局部标号 1。

第 15 行，实现了对局部标号 1 的引用。

3.3.5 外部标号

在 ARM 汇编源文件中，对外部标号的引用是通过伪操作 IMPORT 来实现的。

IMPORT 伪操作用于告诉编译器当前的符号不是在本源文件中定义的，而是在其他源文件中定义的，在本源文件中可能引用该符号。

例 1：IMPORT |Image$$RO$$Base|。告诉编译器|Image$$RO$$Base|是在其他文件中定义的，本文件可能引用该符号。实际上，|Image$$RO$$Base|是编译器产生的一个符号，用于表示 RO 段的结束地址。

此外，在 ARM 汇编源文件中可以用 EXPORT 伪操作声明一个外部标号，即当前标号是在本源文件中定义的，在其他文件中可能会被引用。

例 2：IMPORT RdNF2SDRAM。告诉编译器 RdNF2SDRAM 是在其他文件中定义的，本文件可能引用该标号。

例 3：EXPORT StartPointAfterSleepWakeUp。告诉编译器 StartPointAfterSleepWakeUp 是在本文件中定义的，其他文件可能引用该标号。

3.3.6 文件包含

在编写汇编源文件时，通常将常量定义放在一个文件中，该文件的后缀为.inc，如用 EQU 定义的外设地址，类似于 C 语言中用 include 包含头文件。然后，通过使用 GET 伪操作将其包含到本源文件中。

例：GET 2440addr.inc 将 2440addr.inc 文件包含到文件中，文件 2440addr.inc 中定义了 S3C2440 的寄存器地址。

注意：汇编语言中被包含的文件常以.inc 结尾。

3.4 用 AXD 调试 ARM 汇编程序实验

在前面的章节中介绍了 ARM 指令集中的常用指令和 ARM 汇编语言基础知识，读者可在一定程度上了解 ARM 汇编程序设计的基本要领。在本节中，结合一个具体的 ARM 汇编语言源程序，给读者展示如何利用 AXD 进行程序的仿真调试。

经过本节的学习，读者应该掌握用 AXD 调试器进行程序调试的一些实用技巧，如查看寄存器的值、查看内存单元的值、单步执行等，通过在程序调试过程中反映出的问题，加深对 ARM 汇编指令的理解。

3.4.1 建立工程并添加源文件

单击：开始\程序\ARM Developer Suite v1.2\CodeWarrior for ARM Developer Suite，启动 Metrowerks CodeWarrior，启动 ADS 1.2 过程如图 3-41 所示。

单击 File\New 即可弹出“New（新建工程）”对话框，如图 3-42 所示。选择“ARM Executable Image”（ARM 可执行映像文件），输入工程名（例如，test），选择保存路径即可。

用户可以在刚才建立的工程下建立一个文本文件，以便输入用户程序，单击“New Text

File”按钮，如图 3-43 所示。然后，在新建的文件中输入源程序，保存为 test.s，test.s 的内容如图 3-44 所示，细心的读者可能已经发现，test.s 中的内容是本章前面的例题所讲解的指令。通过本节的学习，希望读者能够更好地掌握这些基本的指令。

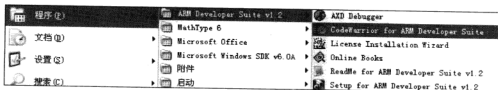


图 3-41　启动 ADS 1.2 过程

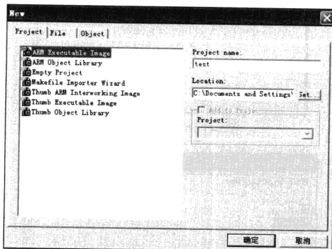


图 3-42　“New（新建工程）”对话框

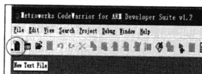
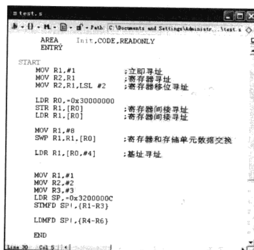


图 3-43　新建源文件



```
test.s
        AREA    Init,CODE,READONLY
        ENTRY

START
        MOV R1,#1              ;立即寻址
        MOV R2,R1              ;寄存器寻址
        MOV R2,R1,LSL #2       ;寄存器移位寻址

        LDR R0,=0x30000000
        STR R1,[R0]            ;寄存器间接寻址
        LDR R1,[R0]            ;寄存器间接寻址

        MOV R1,#8
        SWP R1,R1,[R0]         ;寄存器和存储单元数据交换

        LDR R1,[R0,#4]         ;基址寻址


        MOV R1,#1
        MOV R2,#2
        MOV R3,#3
        LDR SP,=0x3200000C
        STMFD SP!,{R1-R3}

        LDMFD SP!,{R4-R6}

        END
```

图 3-44　test.s 的内容

在工程窗口中单击鼠标右键，会弹出一个悬浮的菜单，选择“Add Files”项，如图 3-45 所示，即可弹出 Select files to add 窗口，选择相应的源文件，单击“打开”按钮即可，如图 3-46 所示。



图 3-45　添加 test.s 到工程

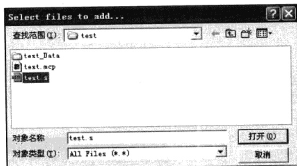


图 3-46　选择“test.s”

3.4.2　工程的设置

在工程窗口中，单击“DebugRel Settings”按钮，如图 3-47 所示，即可弹出“DebugRel Settings”对话框，选择“ARM Assembler”，在“Architecture or Processor”的下拉列表框中选择“ARM920T”，如图 3-48 所示。

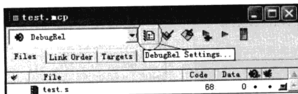


图 3-47　单击“DebugRel Settings”按钮

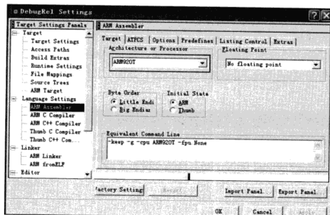


图 3-48　“DebugRel Settings”对话框

3.4.3　编译源文件

在工程窗口中单击“Make”按钮，如图 3-49 所示，即可实现对源文件的编译。

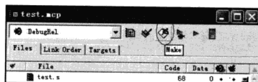


图 3-49　单击“Make”按钮

3.4.4　启动 AXD 调试器

在工程窗口单击“Debug”按钮，如图 3-50 所示，即打开 AXD 调试器。当用户第一次打开 AXD 调试器时，可能会出现如图 3-51 所示的错误，这主要是由于 AXD 调试器设置不正确引起的。

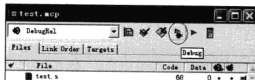


图 3-50　单击“Debug”按钮

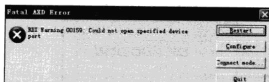


图 3-51　启动 AXD 常见错误

解决如图 3-51 所示的错误的方法是：单击“Configure”，弹出如图 3-52 所示的对话框，单击“Add”按钮，弹出“打开”对话框，选择 ADS 1.2 安装目录，在 Bin 文件夹下选择 ARMulate.dll 文件，单击“打开”按钮即可。例如，假设 ADS 1.2 安装在 D 盘 Program Files 文件夹下，则依次在 D:\Program Files\ARM\ADSv1_2\Bin 文件夹下找到 ARMulate.dll 文件，如图 3-53 所示。

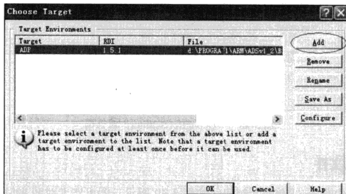


图 3-52　单击“Add”按钮

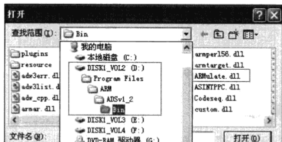


图 3-53　选择 ARMulate.dll 文件

然后，在如图 3-52 所示的“Choose Target”对话框中单击“Configure”按钮，此时会弹出“ARMulator Configuration”对话框，在“Variant”的下拉列表框中选择“ARM920T”，如图 3-54 所示。

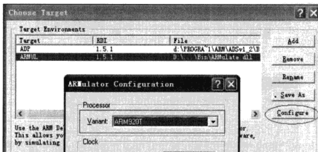


图 3-54　选择“ARM920T”

在 AXD 调试器主界面，选择“File”菜单，然后选择“Load Image”项，如图 3-55 所示，即可打开“Load Image”对话框，选择相应的映像文件即可，如图 3-56 所示。

最后，进入调试状态，AXD 调试器窗口如图 3-57 所示。

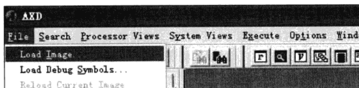


图 3-55　装载映像文件

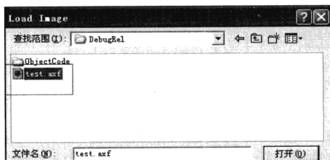


图 3-56　选择映像文件

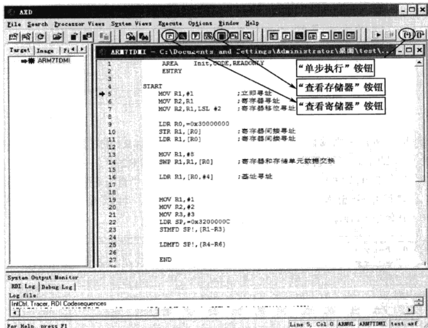


图 3-57　AXD 调试器窗口

3.4.5 手把手调试汇编程序

在图 3-57 中，单击“查看寄存器”按钮，可以查看寄存器中的数值；单击“查看存储器”按钮，可以查看存储单元的数据；单击“单步执行”按钮，即可执行第一条指令。该条指令的功能是将立即数 1 加载到寄存器 R1 中，执行完该条指令后，可以单击“查看寄存器”按钮，此时可以在 AXD 调试器左边栏中看到寄存器 R1 的值变为 0x00000001。注意，当寄存器的值改变时，会以红色字体显示，如图 3-58 所示。注意，此时 AXD 调试器中左边蓝色小箭头指向了第 6 行（注意，以下对程序中第几行的描述都是基于图 3-57 所示的程序），这是程序中下一条将要执行的指令。

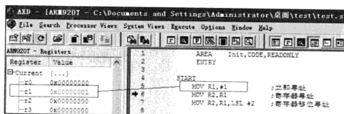


图 3-58 “小箭头”指示下一条将要执行的指令

单击“单步执行”按钮（或者按键盘上的 F10 键），执行指令 MOV R2,R1，这是寄存器寻址，即将寄存器 R1 的值传送到寄存器 R2 中，因此，执行完这条指令后，寄存器 R2 的值应为 1，如图 3-59 所示，寄存器 R2 的值为 0x00000001。

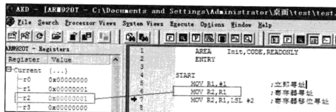


图 3-59 第 2 条指令执行效果

第 7 行是一条寄存器移位寻址的指令，此时先将寄存器 R1 的值左移 2 位，然后传送到寄存器 R2 中，即 1 左移 2 位是 4，然后将 4 传送到寄存器 R2 中。

第 9 行指令将 0x30000000 加载到寄存器 R0 中。单击“查看存储器”按钮，在弹出的“Memory”对话框中输入地址 0x30000000，按回车键，可以看到窗口中显示的是从 0x30000000 开始的内存单元中的数据，如图 3-60 所示。

ARM920T - Memory Start addr 0x30000000

Tab1 - Hex - No prefix | Tab2 - Hex - No prefix | Tab3 - Hex - No prefix | Tab4 - Hex - No prefix |

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	
0x30000000	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8
0x30000010	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8
0x30000020	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8
0x30000030	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8
0x30000040	10	00	FF	E7	00	E8	00	E8	10	00	FF	E7	00	E8	00	E8

图 3-60 查看 0x30000000 内存单元

第 10 行将寄存器 R1 的值（此时是 0x00000001）存储到 R0 所指向的内存地址处，即将 0x00000001 存储到地址 0x30000000 处。执行完该条指令后可以看到，内存单元 0x30000000 处的值已经变成以字节为单位（红色字体）显示（当存储单元的数据改变时会以红色字体显示，这样便于用户观察存储单元的数据变化），如图 3-61 所示。因为 ARM 存储器默认是小端存储方式，因此数据低位存放在存储单元的地址值处。

ARM920T - Memory Start addr 0x30000000

Tab1 - Hex - No prefix | Tab2 - Hex - No prefix | Tab3 - Hex

Address	0	1	2	3	4	5	6	7
0x30000000	01	00	00	00	00	E8	00	E8
0x30000010	10	00	FF	E7	00	E8	00	E8
0x30000020	10	00	FF	E7	00	E8	00	E8
0x30000030	10	00	FF	E7	00	E8	00	E8
0x30000040	10	00	FF	E7	00	E8	00	E8
0x30000050	10	00	FF	E7	00	E8	00	E8
0x30000060	10	00	FF	E7	00	E8	00	E8
0x30000070	10	00	FF	E7	00	E8	00	E8

图 3-61　以字节为单位显示内存单元内容

第 13～14 行程序是为了验证 SWP 指令的功能。SWP 指令的功能是将存储单元中的数据和寄存器中的数据交换，第 13 行将指令将立即数 8 加载到寄存器 R1 中，而此时 R0 的值为 0x30000000，此时存储单元 0x30000000 中的数据是 0x00000001，如图 3-61 所示。因此执行完 SWP R1,R1,[R0]后，寄存器 R1 的值应该是 0x00000001，存储单元 0x30000000 中的数据应该是 0x00000008，如图 3-62 所示。

ARM920T - Registers

Register	Value
Current	{...}
r0	0x30000000
r1	0x00000001
r2	0x00000004
r3	0x00000000
r4	0x00000000
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000

```
5        MOV R1,#1
6        MOV R2,R1
7        MOV R2,R1,LSL #
8
9        LDR R0,=0x30000
10       STR R1,[R0]
11       LDR R1,[R0]
12
13       MOV R1,#8
14       SWP R1,R1,[R0]
15
16       LDR R1,[R0,#4]
17
18
19       MOV R1,#1
```

ARM920T - Memory Start addr 0x30000000

Tab1 - Hex - No prefix | Tab2 - Hex - No prefix | Tab3 - Hex - No prefix

Address	0	1	2	3	4	5	6	7	8	9
0x30000000	08	00	00	00	00	E8	00	E8	10	00
0x30000010	10	00	FF	E7	00	E8	00	E8	10	00
0x30000020	10	00	FF	E7	00	E8	00	E8	10	00

图 3-62　SWP 指令执行结果

第 19～21 行执行完后，寄存器 R1 的值为 1，寄存器 R2 的值为 2，寄存器 R3 的值为 3。

第 22 行将堆栈指针指向 0x3200000C 处，此时在 Memory 窗口中输入 0x3200000C，可以查看存储单元 0x3200000C 处的值，如图 3-63 所示，在左边栏中可以看到堆栈指针 R13（SP）的值为 0x3200000C。

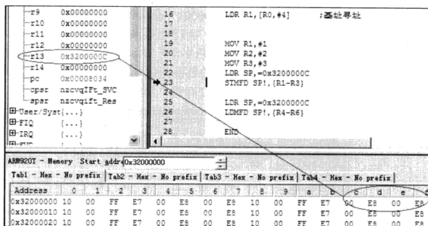


图 3-63　堆栈指针的位置

第 23 行是 STMFD SP!，{R1-R3}。该指令的指令执行过程：首先，堆栈指针 SP 减 4（因为 ARM 指令是 32 位的，一次传送 4 个字节），即此时 SP=0x30000008，将寄存器 R3 中的数据 3 入栈；然后，堆栈指针 SP 再减 4，即此时 SP=0x30000004，将寄存器 R2 中的数据 2 入栈；最后，堆栈指针 SP 再减 4，即此时 SP=0x30000000，将寄存器 R1 中的数据 1 入栈。由于指令中 SP 后面有“！”，表示将最后堆栈指针更新，即此时 SP=0x30000000（读者可以尝试着将 SP 后面的“！”去掉，如果去掉，SP 的值就不会更新）。由上面的分析知，堆栈指针始终指向最后一个入栈的数据（注意，结合 ARM 堆栈是满递减的含义加以理解）。此时可以看到堆栈中数据的变化情况，如图 3-64 所示，可以看到，此时寄存器 R13 的值为 0x32000000，并且 R3 先入栈，R1 最后入栈。

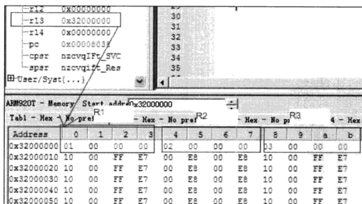


图 3-64　数据入栈示意图

第 26 行是 LDMFD SP!，{R4-R6}。该指令的含义：数据出栈，放入 R4～R6 寄存器中，注意，“！”说明最后堆栈指针更新。指令执行过程：首先，0x00000001 出栈保存到寄存器 R4 中，堆栈指针 SP 加 4（因为 ARM 指令是 32 位的，一次传送 4 个字节），此时 SP=0x30000004；然后，0x00000002 出栈保存到寄存器 R5 中，堆栈指针 SP 再加 4，此时

SP=0x30000008；最后，0x00000003 出栈保存到寄存器 R6 中，堆栈指针再加 4，此时 SP=0x3000000C。最后程序执行结果如图 3-65 所示。

总结：结合一个简单的汇编语言程序，向读者展示了用 AXD 调试器进行程序调试的基本方法，同时复习了前面讲解的基础知识。读者可以结合自己的实际情况有针对性地练习，不用机械地记忆指令，可以边调试边预测指令执行的过程，然后通过调试结果来验证自己的理解是否正确。相信经过一定的练习，读者可以很快掌握 ARM 汇编方法。

Register	Value
Current	{...}
r0	0x30000000
r1	0x00000001
r2	0x00000002
r3	0x00000003
r4	0x00000001
r5	0x00000002
r6	0x00000003
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000

图 3-65 数据出栈后的情形

3.5 常用汇编语言程序子模块实例分析

在前面的讲解中把重点放在了具体指令的执行过程中，目的是使读者理解指令的执行过程。但是，学习汇编更重要的是掌握一些常用的程序模块，通过程序模块的学习进一步加深对 ARM 指令的综合运用能力。本节将给出部分常用汇编程序模块，读者可以有选择地阅读。

3.5.1 特殊功能寄存器的访问

ARM 处理器 I/O 和内存采用统一编址的方式（参见本章扩展阅读部分）。三星公司 ARM9 处理器 S3C2440 的特殊功能寄存器地址范围为 0x48000000～0x5FFFFFF。其中，看门狗控制寄存器 WTCON 的地址是 0x53000000，关闭看门狗定时器的方法是，向看门狗控制寄存器 WTCON 写入 0。因此，可以采用如下方式关闭看门狗定时器。

```
1    WTCON   EQU   0x53000000
2         ldr  r0,=WTCON
3         mov r1, #0
4         str   r1,[r0]
```

第 1 行通过 EQU 伪操作定义了一个常量 WTCON，其值为 0x53000000。

第 2 行用 ldr 伪指令将 0x53000000 加载到寄存器 r0 中，即此时 r0 指向了看门狗定时器控制寄存器 WTCON。

第 3 行将 0 传送到寄存器 r1。

第 4 行将 0 存储到寄存器 r0 指向的地址处，由于此时 r0 指向了看门狗定时器控制寄存器，因此达到了将看门狗定时器控制寄存器清零的目的，即关闭了看门狗。

总结：在启动代码的编写过程中，大多数对特殊功能寄存器的访问都是通过上述方式实现的，其基本思路是，将初始值写入相应的控制寄存器就可以实现对相关硬件的初始化。例如，初始化 SDRAM 时，是根据具体的 SDRAM 芯片将对应的参数值写入相应的存储器控制寄存器中实现的。

3.5.2 内存数据复制

在实现 NAND FLASH 启动过程中需要将代码搬移到 SDRAM 中，通过后序寻址（参见本章扩展阅读部分）可以很方便地实现数据的搬移。下面的例子给出了代码搬移的基本原理。

例 1：假设 r1 为指向源数据块的起始地址，r2 指向源数据块的结束地址，r3 指向目的数据块的起始地址，如图 3-66 所示。

实现数据搬移的代码如下：

```
1    loop    ldr  r0, [r1], #4
2            str  r0, [r3], #4
3            cmp r1, r2
4            bcc    loop
```

第 1 行指令的执行过程：以 r1 指向的存储单元中取出一个字（4 个字节），将其加载到寄存器 r0 中，同时寄存器 r1 的值自动加 4，然后保存到寄存器 r1 中，即此时 r1 指向下一个存储单元。

第 2 行将寄存器 r0 的值存储到寄存器 r3 指向的地址处，然后寄存器 r3 的值自动加 4 保存到寄存器 r3 中，即此时 r3 指向下一个地址处。

第 3 行比较 r1 和 r2 的大小。

第 4 行是条件执行的跳转指令，当 r1 小于 r2 时跳转到第 1 行处执行，即如果数据没有搬完，则接着搬，直到搬完为止。

更形象的理解：寄存器 r0 的作用就像一辆车，先从 r1 指向的地址处将数据取出来装到车上，然后将数据“运输”到 r3 指向的地址处，直到将所有数据“运输”完为止，图 3-67 形象地展示了这一过程。

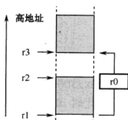


图 3-66　数据搬移示意图

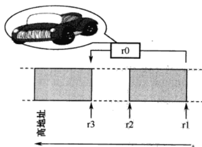


图 3-67　数据搬移形象化展示

3.5.3 批量加载与存储

在启动代码中，会将与外设有关的参数加载到 ARM 处理器与之相对应的控制寄存器中。当控制寄存器数量较多时，可以使用批量加载与存储指令来实现。如果说前面例子中寄存器 r0 的作用就像一辆汽车，那么在下面这个例子中，寄存器 r1～r13 更像是一辆火车，将数据一次性“运输”到目标地址。

例：下面是启动代码中初始化 SDRAM 的一段代码。SMRDATA 是在内存中定义的一个数据表，占据 13 个字（52 个字节）的空间，用来存放与存储器控制器相关的 13 个寄存器的初始化值；BWSCON 是 S3C2440 处理器存储器控制器的起始地址。

```
1    adrl    r0, SMRDATA
2    ldmia   r0, {r1-r13}
3    ldr     r0, =BWSCON
4    stmia   r0, {r1-r13}
```

第 1 行用 adrl 伪指令加载数据表的首地址到寄存器 r0。

第 2 行用批量加载指令 ldmia 将 r0 指向的数据表中的 13 个参数(每个参数占 4 个字节)加载到寄存器组 r1～r13 中。

第 3 行用 ldr 伪指令将 S3C2440 处理器存储器控制器的起始地址加载到寄存器 r0 中。

第 4 行用批量存储指令 stmia 将寄存器组 r1～r13 中的数据依次存储到 r0 指向的 13 个存储器控制寄存器中。

图 3-68 展现了初始化 SDRAM 过程。

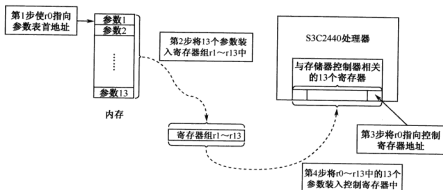


图 3-68　初始化 SDRAM 过程

细心的读者可能会发现，同样是将 32 位地址加载到寄存器 r0 中，第 1 行加载数据表的首地址到寄存器 r0 时用的是 adrl 伪指令，而第 3 行加载 S3C2440 处理器中存储器控制寄存器的首地址时用的是 ldr 伪指令。可以将第 1 行的 adrl 替换成 ldr 吗？为什么？在第 7 章分析启动代码时，读者会有一个清晰的认识。

3.5.4　堆栈操作

ARM 处理器有 7 种工作模式，各个模式都有自己的堆栈。在系统启动时，启动代码需要初始化各模式的堆栈。初始化堆栈采取的方法是：首先切换到相应的处理器工作模式，然后对该模式下的堆栈指针 SP 赋值。

例：堆栈初始化过程分析，示意代码如下。

```
1    FIQMODE     EQU    0x11
2    IRQMODE     EQU    0x12
```

```
3     SVCMODE          EQU     0x13
4     MODEMASK         EQU     0x1f
5     NOINT            EQU     0xc0
6     _STACK_BASEADDRESS EQU 0x33ff8000

7     FIQStack      EQU    (_STACK_BASEADDRESS-0x0)      ;0x33ff8000 ~
8     IRQStack      EQU    (_STACK_BASEADDRESS-0x1000);0x33ff7000 ~
9     SVCStack      EQU    (_STACK_BASEADDRESS-0x2800);0x33ff5800 ~

10    InitStacks
11        mrs   r0,cpsr
12        bic   r0,r0,#MODEMASK
13        orr   r1,r0,#IRQMODE|NOINT
14        msr   cpsr_cxsf,r1          ;IRQMode
15        ldr   sp,=IRQStack          ; IRQStack=0x33FF7000

16        orr   r1,r0,#FIQMODE|NOINT
17        msr   cpsr_c,r1             ;FIQMode
18        ldr   sp,=FIQStack          ; FIQStack=0x33FF8000

19        bic   r0,r0,#MODEMASK|NOINT
20        orr   r1,r0,#SVCMODE
21        msr   cpsr_cxsf,r1          ;SVCMode
22        ldr   sp,=SVCStack          ; SVCStack=0x33FF5800
23        mov   pc,lr
```

第 1～5 行用 EQU 定义了 5 个常量，回顾 1.2.3 节讲述的程序状态寄存器 CPSR 中模式控制位的含义，可以很容易理解上述 4 个常量的含义。

第 6 行定义了一个常量，该常量代表了内存中的地址 0x33ff8000（TQ2440 开发板的 SDRAM 地址范围是 0x30000000～0x34000000）。

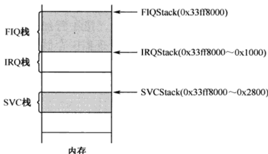


图 3-69　堆栈空间

第 7～9 行，分配快速中断模式（FIQ）、外部中断模式（IRQ）和管理模式（SVC）下的堆栈空间，如图 3-69 所示。

第 11 行 mrs 是读程序状态寄存器指令，将程序状态寄存器 CPSR 中的内容读入到 r0 中。

第 12 行 bic 是位清零指令。此时，r0 中存放的是程序状态寄存器 CPSR 中的值。因此，该条指令将 CPSR 中模式控制位（低 5 位）清零。

第 13 行通过或指令 orr 对 r0 的模式控制位赋值（对模式控制位进行赋值即可实现处理器工作模式的切换），将结果保存到寄存器 R1 中。

第 14 行的 msr 是写程序状态寄存器指令，将寄存器 R1 中的值写入到程序状态寄存器

CPSR 中。第 11～14 行指令实现了将处理器工作模式切换到外部中断模式（IRQ），采用的方法是：读—修改—写的方式实现。

第 15 行使用 ldr 伪指令加载外部中断模式（IRQ）堆栈起始地址到外部中断模式（IRQ）下的堆栈指针寄存器 SP 中。到此，完成了对外部中断模式（IRQ）堆栈的初始化。

第 16～18 行完成了对快速中断模式（FIQ）堆栈的初始化。

第 19～22 行完成了对管理模式（SVC）堆栈的初始化。

3.5.5 实现查表功能

实现查表操作的基本思路是：首先找到表的首地址，然后找到数据距离首地址的偏移量，将其余表的首地址相加即可得到要查找的数据的地址。

例：汇编程序实现查表操作。

```
1           mov r9,#4
2           ldr r8,=DATATABLE
3           ldr r8,[r8,r9,lsl #2]
4      DATATABLE dcd 0x10,0x20,0x30,0x40,0x50
                 dcd 0x60,0x70,0x80,0x90,0xa0
```

第 2 行通过 ldr 伪指令将数据表的首地址（注意，标号 DATATABLE 代表的是一个地址）加载到寄存器 R8 中。

第 3 行是 ldr r8,[r8,r9,lsl #2]。该条指令的执行过程：R9 左移 2 位然后与 R8 相加，将相加的结果加载到寄存器 R8 中。因为 dcd 伪操作是以字（4 个字节）为单位进行分配内存单元的，因此 R9 左移 2 位恰好是 4 字节对齐的。

3.6 本章小结

本章对 ARM 指令集中的指令有选择地进行了讲解，同时结合 AXD 调试器向读者展示了程序调试的基本方法。此外，还对 ARM 汇编程序的基本结构进行了分析，对 ARM 汇编中的常用汇编语言子程序进行了剖析。

3.7 扩展阅读之内存和 I/O 地址、前序寻址和后序寻址

内存和 I/O 地址之间有什么关系呢？什么是前序寻址和后序寻址呢？可能很多初学者都会碰到类似的问题。本节将针对上述问题向读者进行阐述。

1. 内存和 I/O 地址简介

ARM 的寻址空间是线性地址空间，因为地址线是 32 位的，因此，最大寻址空间为 232=4GB，但这并不是说可用的内存是 4GB，实际的内存是由处理器实际引出的地址线的条数决定的。以三星公司 ARM9 处理器 S3C2440 为例，实际引出的地址线为 27 根，也就是说实际可用的内存空间是 227=128MB（128MB 是对一个 BANK 而言，通过控制不同的

BANK 片选信号 nGCS 可以实现对 1GB 的地址空间的访问)。

所谓的 I/O 端口的编址方式是指 I/O 端口地址的安排方式，常见的 I/O 端口的编址方式有两种：统一编址（存储器映射编址或内存映射编址）和独立编址。

- 统一编址

统一编址是指将 I/O 端口地址和内存地址统一安排，即 I/O 端口和内存单元地址在同一个地址空间中，如 ARM 处理器就是这种情况。这种编址方式的优点是，可以采用内存访问指令来访问 I/O 端口，访问 I/O 端口就跟访问内存单元一样，不需要额外的 I/O 指令。缺点是外设占用了部分内存空间。例如，三星公司 ARM9 处理器 S3C2440 的特殊功能寄存器地址范围为 0x48000000～0x5FFFFFF。

- 独立编址

独立编址是指将 I/O 地址空间和内存地址空间分开编址，各自的地址空间相互独立。优点是 I/O 单元不会占用内存空间；缺点是需要提供专门的 I/O 访问指令，还需要 I/O 地址译码电路。例如，Intel x86 系列处理器就是采用独立编址的方式，I/O 端口和内存单元分开编址，I/O 端口有自己独立的地址空间，同时也有 I/O 地址译码电路和专用的 I/O 访问指令。

2．前序寻址和后序寻址（Pre or Post Indexed Addressing）

例 1：前序寻址：STR r0, [r1, #8]。

指令执行过程分析如下。

第 1 步：计算操作数的实际地址为 r1+8。

第 2 步：从地址 r1+8 处取出数据并将其加载到寄存器 r0 中，如图 3-70 所示。

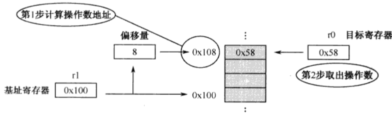


图 3-70　前序寻址示意图

例 2：后序寻址：STR r0, [r1], #8。

指令执行过程分析如下。

第 1 步：从地址 r1 处取出数据并将其加载到寄存器 r0 中。

第 2 步：自动更新基址寄存器 r1 的值 r1=r1+8，如图 3-71 所示。

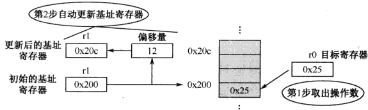


图 3-71　后序寻址示意图

第4章 ARM C 语言基础

本书始终秉承“提供机制而非策略”的理念，本章并不是对C语言的基础知识进行全面系统性的介绍，更注重的是在ARM开发过程中如何使用C语言以及需要注意的问题。

4.1 数据类型基础

ARM 编译器支持整数型和浮点型数据，在本书所有实验中仅仅用到的数据类型有char、int两种。

- char 表示1个字节的数据，长度为8位。
- int 表示1个字数据，长度为32位。

怎么去理解数据类型呢？可能很多人在小时候有自己动手制作雪糕的经历，把绿豆汤加点奶油拌匀后倒在制作雪糕的模子里面，然后把模子放到冰箱里冻几个小时，最后取出模子，雪糕就做好了，而且雪糕的形状跟模子的形状一样。其实，这个“模子”就像这里的数据类型，定义变量就像在内存里面“冻雪糕”。

现在就来尝试着理解：当写 int var=3 的时候，其实是在内存里面“冻结”了一个跟 int 这个“模子”一样大小的内存片，这里是4个字节，并且向这个内存片里面写入初值3，那么以后怎么找到这个内存片呢？给它起个名字 var，因此在程序里想改变这个内存片值时就可以对 var 进行赋值。因此，用其他数据类型（如 short、long、float 等）定义变量是一样的道理，都是在内存里面“冻雪糕”，只不过不同的“模子”冻出来的“雪糕”不一样而已。

此外，对于各种数据类型具体大小，读者不必强记，C语言中有专门用于测量这个“模子”大小的关键字 sizeof，虽然这个关键字“自古以来被误认为是个函数”，例如，sizeof（int）计算 int 型数据所占的字节数。

4.1.1 用 typedef 和#define 定义类型

常用 typedef 声明新的类型名来代替已有的类型名，这主要是便于移植。例如，在程序中经常会看到如下语句：

```
1    typedef unsigned int U32
2    ……
3    U32 var = 3 ;
```

第 1 行用 typedef 声明了一个类型 U32 来代替 unsigned int，这样书写起来也比较方便，从 U32 可以很容易地看出这是一个 unsigned 型数据（前面的 U 表示这个意思），后面的 32 表示这是一个 32 位的数据。

第 3 行用刚才声明的 U32 定义了一个变量 var。

在程序中可以看到用#define 来定义新的数据类型。

```
1    #define U32 unsigned int
2    ……
3    U32 var = 3 ；
```

第 1 行用#define 定义了一个类型 U32 来代替 unsigned int。

第 3 行用刚才声明的新类型 U32 定义了一个变量 var。

以上两种类型定义方法的不同之处在于 typedef 是在编译阶段处理的，而#define 是在预处理阶段处理的。

4.1.2 用 signed 和 unsigned 修饰数据类型

关键字 signed 和 unsigned 常作为数据类型修饰符加在数据类型的前面，如 unsigned int。signed 称为有符号型，unsigned 称为无符号型。

例：signed int 和 unsigned int。

分析：signed int 表示有符号整型，一个 int 型数据是 32 位，前面加 signed 修饰符后，最高位用做符号位（0 表示正数，1 表示负数），剩下的 31 位才是数据的有效位，因此 signed int 能表示的数据范围是-2^{31}～$2^{31}-1$。unsigned int 表示无符号整型，32 为有效数据位，因此一个 unsigned int 表示数据范围是：0～$2^{32}-1$。

4.1.3 volatile 和强制类型转换

C 语言总共有多少个关键字呢？只有 32 个，而 volatile 就是其中的一个，可能部分初级程序员也只知道它的存在而已，恰似“杨家有女初成长，养在深闺人未识”。惊奇吧？为什么学习C语言的时候没有用到volatile关键字呢？因为这个关键字用来修饰变量时表示该变量的值可能被硬件更改，因此每次读取这个变量值的时候要重新从内存中读取这个变量的值，而不是使用保存在寄存器里的备份。

此外，从前文中关于数据类型和“模子”的讨论中可以看到，不同的数据类型长度是不一样的（就像不同模子的大小不一样），因此当操作数的数据类型不一样时需要用到类型转换（当然，C 语言内部有隐式转换规则），将其称为强制类型转换。

例：#define rBWSCON   (*(volatile unsigned *)0x48000000)。

分析：为了便于理解，可以暂时把 volatile 去掉，因此关键是理解这个定义(*(unsigned *)0x48000000)，0x48000000 仅仅是一个十六进制表示的数据而已，但是前面用(unsigned *)修饰，表示将 0x48000000 强制转换为一个地址指针，即(unsigned *)48000000 指向内存中地址 0x48000000 处，准确地说是指向内存中从 0x48000000 开始的连续的 4 个字节的内存片中（0x48000000～0x48000003）。因此，(unsigned *)48000000 其实就是(unsigned int

*)48000000 的缩写，如图 4-1 所示。

然后，在(unsigned *)48000000 前面再加个*，这里的*是指针运算符（也叫"间接访问"运算符），表示取该内存单元中的数据。

最后再看#define rBWSCON (*(unsigned *)0x48000000)，表示用#define 定义了一个新的类型 rBWSCON，rBWSCON 含义和(*(unsigned *)0x48000000)完全相同，都表示访问内存单元 0x48000000 中的数据。因此，在程序中可以看到下面的语句：rBWSCON = 0x00000003，就相当于(*(unsigned *)0x48000000) = 0x00000003，其实就是向内存单元 48000000 中写入了对应的数据，如图 4-2 所示。

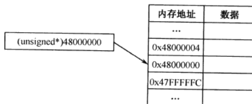


图 4-1 (unsigned *)48000000 实例

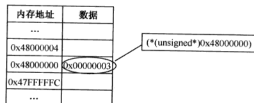


图 4-2 (*(unsigned *)0x48000000)示意图

此外，读者可能已经注意到，在刚开始讨论时，为了讨论方便把 volatile 去掉了，其实 volatile 关键字只是表示每次读写该内存单元中的数据时都要到内存单元处去读取，而不是读取寄存器中的备份值。

4.2 深入理解位运算符和位运算

位运算是指二进制位之间的运算。在嵌入式系统设计中，常常要处理二进制位的问题，如将某个寄存器中的某一位置 1 或置 0，将数据左移 5 位等。本书中常用的位运算符如表 4-1 所示。

表 4-1 位运算符

运 算 符	含 义
&	按位与
|	按位或
~	按位取反
<<	左移
>>	右移

4.2.1 按位与运算符（&）

按位与运算规则：参加运算的两个操作数，每个二进制位进行"与"运算，若两位都是 1，则结果为 1，否则为 0。

例：1001 & 1011 运算过程如图 4-3 所示。

4.2.2 按位或运算符（|）

按位或运算规则：参加运算的两个操作数，每个二进制位进行“或”运算，若两位都是 0，则结果为 0，否则为 1。

例：1000 | 1010 运算过程如图 4-4 所示。

```
  1001
& 1011
------
  1001
```

图 4-3 按位与运算

```
  1000
| 1010
------
  1010
```

图 4-4 按位或运算

4.2.3 按位取反运算符（~）

按位取反运算符用来对一个二进制数按位取反。

例：～1000 表示将 1000 按位取反，得到结果 0111。

4.2.4 左移和右移运算符（<<）、（>>）

左移运算（<<）用来将一个数左移若干位，右移运算（>>）用来将一个数右移若干位。

例 1：假设 val 是个 unsigned char 型数据，对应的二进制数是 10010110，则 val = val << 3，表示将 val 左移 3 位然后赋值给 val，注意在左移过程中，高位移出去后被丢弃，低位补 0。最后，val=10110000。

例 2：假设 val 是个 unsigned char 型数据，对应的二进制数是 10010110，则 val = val >> 3，表示将 val 右移 3 位然后赋值给 val，注意在右移过程中，低位移出去后被丢弃，高位补 0。最后，val=000100101。

4.2.5 位运算应用实例分析

上述位运算符有什么用处呢？一般按位与用来“清零”，按位或用来“置 1”。

例 1：S3C2440 处理器 I/O 端口 PORT B 共有 10 个引脚，可以修改 POTR B 的控制寄存器 GPBCON 中相应的位来实现将不同的引脚设为输入或者输出功能。

```
1    #define rGPBCON     (*(volatile unsigned *)0x56000010)
2    rGPBCON &= ~(3 << 10) ;
3    rGPBCON |= (1<<10) ;
```

第 1 行用#define 定义了 rGPBCON，以下对 rGPBCON 的访问其实就是对内存单元 0x56000010 的访问（准确地说是对 0x56000010～0x56000013 这个内存片的访问），这正是 GPBCON 寄存器的地址，因此对 rGPBCON 的访问就是对控制寄存器 GPBCON 的访问。

第 2 行(3 << 10)得到 00000000000000000000110000000000，然后按位取反得到 11111111 111111111111110011111111 1。rGPBCON &= ~(3 << 10)相当于 rGPBCON =rGPBCON & (~(3 << 10))，即将 rGPBCON 的值与~(3 << 10)按位与。通俗的理解就是将 rGPBCON 中第 10、11 两位清零！

第 3 行 rGPBCON |= (1<<10)相当于 rGPBCON = rGPBCON | (1<<10)，有了前面的分析，这一句很好理解：将第 10、11 位的值赋为 01，此时 GPB5 被设置成输出功能。

读者可能会问：*为什么要用上面的步骤来将 GPBCON 的第 10、11 位的值赋为 01 呢？其实这是在开发中的一小技巧，对硬件寄存器的访问都是采取这种“先与后或”的方式。这样的优点是不会影响寄存器中其他位的设置（与“1”相与值不变，与“0”相或值不变）。*

例 2：下面的例子展示了如何点亮一个 LED。假设当 GPB5 输出低电平时 LED 亮，则点亮 LED 的思路可以总结为：第一，通过配置 GPBCON 寄存器将 GPB5 设置为输出功能；第二，向寄存器 GPBDAT 的第 5 位写入 0（这里就涉及将某一位“清零”），即可在 GPB5 引脚输出低电平，此时 LED 就会点亮了。

下面是控制 LED 亮灭的程序：

```
1    #define rGPBDAT      (*(volatile unsigned *)0x56000014)
2    rGPBDAT &= (~(1 << 5));
3    rGPBDAT |= (1 << 5) ;
```

第 1 行用#define 定义了 rGPBDAT，以下对 rGPBDAT 的访问其实就是对控制寄存器 GPBDAT 的访问。

第 2 行将 GPBDAT 寄存器的第 5 位清零，实现了 GPB5 引脚输出低电平，也就是点亮了 LED。

第 3 行将 GPBDAT 寄存器的第 5 位置 1，实现了 GPB5 引脚输出高电平，也就是关闭了 LED。

总结：在上面的应用实例中也反映了用 ARM 程序来控制 I/O 端口的基本思路：通过程序控制 ARM 的 I/O 端口时，先要找到 I/O 端口的控制寄存器地址（端口的控制寄存器地址是确定的，参见第 3 章扩展阅读部分），通过“先与后或”的方式将相应的端口设置成所需要的功能（如设置成输入、输出功能等），然后向 I/O 端口的数据寄存器写入相应的值即可，一般会用到位运算中的“清零”和“置 1”功能。

4.3 控制结构

编程中更多的时候需要控制语句来实现程序的分支转移和循环等操作，用于控制程序执行流程的语句主要有选择结构和循环结构。

4.3.1 选择结构

选择结构常用的有 if...else 结构和 switch...case 结构。通过选择结构可以方便地控制程序的执行流程，在后面实验部分用到具体的选择结构时再进行具体讲解。

4.3.2 循环结构

C 语言中提供的循环结构有 while、do...while 和 for 循环，但是根据具体的应用场合可能会选择不同的循环结构来实现相应的功能，在后面实验部分用到具体的循环结构时，再

进行具体讲解。

4.4 防止文件重复包含技巧

编译器对 C 语言源程序进行处理大致经过预处理（Preprocess）、编译（Compile）、汇编（Assemble）和链接（Linking）共 4 个步骤最终才生成可执行程序。一般在对源程序进行语法和词法分析之前，先要对程序进行预处理。C 编译器专门提供了部分预处理指令来指示编译器如何对源程序进行预处理，预处理指令以#开始，单独占一行。

这里讲解的#ifndef 和#endif 主要是用在防止头文件重复包含的情况下，这对于模块化开发至关重要。

例：下面是 led.h 文件的内容。

```
1    #ifndef __LED_H__
2    #define __LED_H__
3    extern void Led_Init();
4    extern void Led1_On();
5    extern void Led1_Off();
6    #endif
```

第 1 行用#ifndef 测试__LED_H__是否定义过。如果没有定义，则会执行第 2 行定义__LED_H__，然后依次执行下面的声明语句。

第 3～5 行用 extern 声明了一个外部函数，即函数 Led_Init()是在其他文件中定义的，本文件可能要用到，因此要声明一下。

4.5 ARM 编译器对 C 语言的扩展

ARM 编译器提供了很多对 C 语言扩展的关键字，如__irq、__swi、__asm、__inline 等。初学阶段只需要掌握__irq、__swi 就可以满足正常的开发需求。使用__irq 关键字来定义中断处理函数，当中断发生时，编译器会自动保存相应寄存器的值。使用__swi 来定义软中断。

注意：__irq、__swi 等关键字的前面是两个下画线。

4.5.1 __irq 声明中断处理函数

用__irq 声明中断处理函数的实例如下：

```
1    void __irq Timer0_Isr(void)
2    {
3        /* 这里写定时器 0 的中断处理程序 */
4        rSRCPND   =   1 << 10 ;//清除中断标志位
5        rINTPND   =   1 << 10 ;
6    }
```

第 1 行用关键字__irq 声明了定时器 0 的中断处理函数 Timer0_Isr。

第 3 行在这里写定时器 0 的中断处理函数要执行的功能。

第 4、5 行将定时器 0 中断标志清除。

当然，ARM 处理器处理中断时还需要涉及在汇编语言中注册相应的中断处理函数等知识，但在此读者只需要对中断处理函数的格式有所了解即可，结合本书后面的具体实验可以很快掌握中断处理的全过程。只要把 C 语言中的中断处理函数写正确了，其他问题都会很容易解决。

4.5.2 __swi 声明软中断

软中断的主要功能是：将处理器工作模式切换到管理模式，主要是为了支持操作系统的系统功能调用接口或者也可以使用软中断来实现任务的切换等。但是，在裸机开发中，读者需要了解中断的执行流程以及如何编写一个软中断处理函数。只要能掌握这些基本的功能，相信移植 μC/OS-Ⅱ到 TQ2440 开发板将不再是什么难事。

例：声明软中断的基本形式为__swi(0x20) void ledtest()。

分析：__swi 表示声明函数 ledtest()是个软中断，这样在程序中调用 ledtest()函数时将会产生个软中断，系统执行相应的现场保护，然后执行软中断处理程序，执行完后恢复软中断现场，接着执行用户程序。软中断处理流程如图 4-5 所示，在此省略了很多细节，只是把处理流程展示出来了。

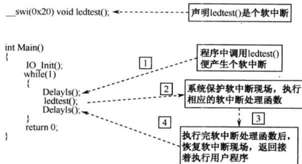


图 4-5 软中断处理流程

4.5.3 __asm 内嵌汇编

通常在 C 程序中需要嵌入汇编代码，这时可以用__asm 关键字来指示编译器下面的代码是用汇编语言写的，请参见 4.5.4 节的例 1。

4.5.4 __inline 定义内联函数

用__inline 关键字定义函数就像在 C 语言中用 define 定义宏一样，用__inline 关键字定义的函数在调用的地方被展开，这主要是为了解决频繁的函数调用开销过大的问题。当然，用__inline 关键字定义函数时，如果函数代码太大，每个调用该函数的地方都会将其展开，

这也会在一定程度上增加代码量，所以一般用__inline 定义的函数代码并不是很大。

常用的情况是在 C 语言中定义开/关中断的函数。

例 1：对 ARM 处理器而言，开中断只需要将当前程序状态寄存器 CPSR 中 I 位清零即可，CPSR 各位的含义如图 4-6 所示的当前程序状态寄存器 CPSR。

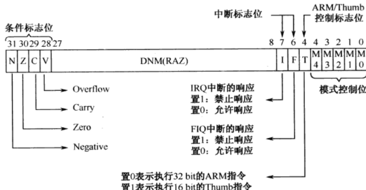


图 4-6　当前程序状态寄存器 CPSR

由于在 C 语言中无法直接访问 CPSR，因此需要在 C 语言中通过嵌入汇编语言来将 CPSR 中 I 位清零，进而实现开中断。此外，访问 CPSR 还需要用专门的程序状态寄存器访问指令 MRS/MSR 来实现。

例 1：用如下函数实现开中断。

```
1    __inline void irq_enable(void)
     {
2        int val;
3        __asm
         {
4            mrs val,cpsr
5            bic val,val,#0x80
6            msr cpsr_c,val
         }
     }
```

第 1 行用__inline 关键字声明了一个内联函数 irq_enable()。

第 2 行定义了一个临时变量，用来保存 cpsr 的值。

第 3 行用__asm 关键字告诉编译器下面的代码是用汇编语言写的。

第 4 行用 mrs 指令将程序状态寄存器 cpsr 中的值读入到 val 中。

第 5 行用 bic 指令将 val 中第 7 位（I 位）清零。

第 6 行用 msr 指令将 val 的值写入到 cpsr 中，此时 I 位已经被清零，即开中断。

例 2：用如下函数实现关中断。

```
1    __inline void irq_disable(void)
```

```
    {
2       int val ;
3       __asm
        {
4           mrs val,cpsr
5           orr val,val,#0x80
6           msr cpsr_c,val
        }
    }
```

第 1 行用__inline 关键字声明了一个内联函数 irq_disable()。

第 2 行定义了一个临时变量，用来保存 cpsr 的值。

第 3 行用__asm 关键字告诉编译器下面的代码是用汇编语言写的。

第 4 行用 mrs 指令将程序状态寄存器 cpsr 中的值读入到 val 中。

第 5 行用 orr 指令将 val 中第 7 位（I 位）置 1。

第 6 行用 msr 指令将 val 的值写入到 cpsr 中，此时 I 位已经被置 1，即关中断。

4.6 本章小结

本章对 ARM C 语言程序开发过程中的基础知识进行了讲解，重点分析了位运算的定义和具体应用实例。此外，还阐述了 ARM 编译器对 C 语言扩展的几个常用关键字__irq、__swi、__asm、__inline，同时给出了编写中断处理函数的一般规范。

4.7 扩展阅读之高速缓存基础知识

程序执行的局部性规律包括时间局部性规律和空间局部性规律。

- 时间局部性规律，即在程序执行的过程中，刚刚被访问的信息可能很快被再次访问。时间局部性规律的典型情况是程序中存在着大量的循环。
- 空间局部性规律，即在程序执行的过程中，那些与被访问的地址相邻近的信息也有可能很快被访问。空间局部性规律的典型情况是程序顺序执行。

按照程序执行的“局部性规律”，程序中的数据或者代码被访问后，该数据和代码以及临近的数据代码近期将被再次访问的概率要远大于近期未被访问的数据或代码被访问的概率。因此，当数据或代码被访问后，被认为是经常被访问的数据和代码，将被存入到一个高速缓存存储器中，当再次访问该数据或者代码时直接从该高速缓存存储器中读取数据或者代码的值，而不是到内存中重新读取。这个高速缓存存储器就是高速缓存，即 Cache。计算机系统中的存储器层次结构（Memory Hierarchy）如图 4-7 所示。

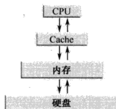


图 4-7 存储器层次结构

高速缓存 Cache 位于内存与 CPU 之间，由 SRAM 组成，容量比较小但速度比主存高得多，接近于 CPU 的速度。采用高速缓存技术主要是解决 CPU 和内存之间速度不匹配的矛盾，尽可能提高处理器的利用率。采用高速缓存技术的处理器已经相当普遍，有的处理器还采用多个高速缓存，如数据高速缓存 Dcache 和指令高速缓存 Icache 等，可尽可能提高系统性能，三星公司 ARM9 处理器 S3C2440 就是这种情况。有了高速缓存 Cache 后，CPU 对内存单元中的数据访问流程如图 4-8 所示。此时，CPU 对内存单元中的数据访问流程是：当 CPU 要读取某个内存单元中的数据时，先到高速缓存中查找，如果找到，则直接读取数据，此时称高速缓存命中；若数据不在高速缓存中，此时称高速缓存不命中，这时需要到内存单元中读取数据，然后将数据及其在内存单元中的地址一起存放在高速缓存中，以便于下次访问该数据时可以从高速缓存中直接读取。

按照结构不同，高速缓存主要有以下 3 种。

- 全关联式高速缓存。
- 直接对应式高速缓存。
- 多组关联式高速缓存。

下面以全关联式高速缓存为例简要讲解其工作原理。全关联式高速缓存由两部分组成，即地址部分（称为标签）和数据部分，如图 4-9 所示（假设地址是 32 位，字长是 32 位，即一次能访问 4 个字节）。

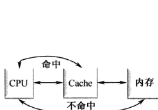


图 4-8　数据访问流程

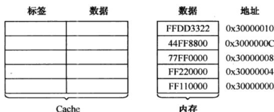


图 4-9　全关联式高速缓存

假设 CPU 要读取内存单元 0x30000004 中的数据，首先用该地址查找 Cache，不命中。将数据 FF220000 读入处理器，同时将地址 0x30000004 写入 Cache 标签字段，对应数据 FF220000 写入 Cache 数据字段；随后 CPU 又要读取内存单元 0x3000000C 中的数据，首先用该地址查找 Cache，不命中。将数据 44FF8800 读入处理器，同时将地址 0x3000000C 写入 Cache 标签字段，对应数据 44FF8800 写入 Cache 数据字段，如图 4-10 所示。



图 4-10　高速缓存不命中的情况

当 CPU 在此读取内存单元 0x30000004 中的数据时，查找高速缓存，此时 0x30000004 中的数据已经位于高速缓存，称为高速缓存命中，此时直接读取高速缓存中的数据即可。但这种情况就有可能导致读取错误的数据，考虑这种情况，假设 0x30000004 正好是个硬件寄存器，有可能在程序执行过程中，系统硬件将 0x30000004 的值改变了。这时，就应该从内存单元 0x30000004 中读取数据，而不是从高速缓存中读取它的备份。

当然，当程序执行一段时间后，高速缓存可能会被填充满。此时，还会涉及高速缓存的数据一致性问题，有兴趣的读者可以参阅相关资料进行学习。

第 5 章

ARM 汇编语言和 C 语言混合编程基础

ARM 体系结构支持 C 和汇编语言的混合编程。在 C 语言中可以调用汇编语言中的子程序，汇编语言也可以调用 C 语言中的子程序。C 语言程序结构清晰，但有些功能是 C 语言无法实现的，如无法用 C 语言实现开/关中断。而这恰恰是汇编语言的优点，所以为了更好地实现程序功能，有时候采取混合编程的方式。例如，当从 NAND FLASH 启动时，需要将代码搬移到 SDRAM 中，这时就需要在汇编语言中调用 C 语言中对 NANF FLASH 的操作函数来实现。

此外，关于混合编程，初学者大可不必为了复杂的参数传递花费太多的时间，只需要掌握基本的几个规则即可。经过一段时间的练习后，就会慢慢掌握并能熟练运用这些规则。

5.1 一个混合编程实例的实现

既然是两种语言程序间的相互调用，这就涉及参数的传递问题。解决好 C 语言程序和汇编语言程序间参数传递和返回值传递的问题是实现混合编程的关键，而 APCS（ARM Process Call Standard）正是定义了一系列的规则来解决上述问题。

下面通过一个实例分析混合编程问题。

本例展示了如何实现 C 语言和汇编语言混合编程的问题，该工程中文件的总体布局如图 5-1 所示。

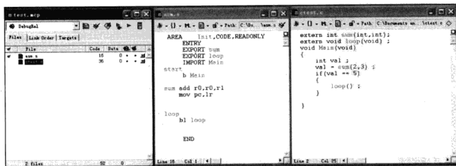


图 5-1　混合编程实例分析

新建一个工程，建立两个源文件：汇编语言源文件 asm.s 和 C 语言源文件 ctest.c。

- 汇编语言源文件 asm.s 的内容如下：

```
1      AREA    Init,CODE,READONLY
2      ENTRY
3      EXPORT sum
4      EXPORT loop
5      IMPORT Main
6  start
7      b Main
8      sum add r0,r0,r1
9      mov pc,lr
10  loop
11      bl loop
12      END
```

第 1、2、12 行是 ARM 汇编程序的基本结构，用 AREA 声明了一个 Init 段，ENTRY 指定了程序入口点，END 指定了汇编程序的结束。请注意：这几行都不能顶格书写。

第 3、4 行用 EXPORT 声明了一个外部标号 sum 和 loop，其实就是在 C 语言中要引用 sum 和 loop，所以要在汇编语言文件中用 EXPORT 将其声明。

第 5 行用 IMPORT 声明了在 C 语言中定义的函数 Main，在汇编语言中调用 C 语言中的函数或者全局变量要用 IMPORT 伪操作在汇编语言文件中声明，否则编译器会报错。

第 6 行定义了一个标号，应顶格书写。

第 7 行用 b 跳转指令，实现程序跳转到 C 语言程序 Main 处。

第 8、9 行给出了书写汇编语言子程序的范例，先顶格写函数名，如 sum，然后写函数的内容，如 add r0，r0，r1，最后通过 mov pc，lr 指令即可实现返回，因为 lr 寄存器中保存了程序返回时的地址。

第 10、11 行定义了一个死循环。

- C 语言源文件 ctest.c 的内容如下：

```
1  extern int sum(int,int) ;
2  extern void loop(void) ;
3  void Main(void)
   {
4      int val ;
5      val = sum(2,3) ;
6      if(val == 5)
       {
7          loop() ;
       }
   }
```

第 1 行用 extern 关键字声明了一个外部函数 sum，这个函数就是在汇编语言文件中定义的。相信很多初学者会有这样的疑问：在汇编语言中定义 sum 时并没有指定函数返回值的类型是 int，也没有指定函数需要两个参数，并且参数的类型是 int，这里怎么声明为 extern

int sum(int,int)呢？这就涉及了 APCS 规则，观察在汇编语言中定义的 sum，函数体部分是 add r0，r0，r1，那么 r0 和 r1 的值是多少呢？APCS 规则是，当 C 语言程序调用汇编语言程序时，寄存器 r0～r4 用来传递函数的参数，链接寄存器 r14（lr）用来保存程序的返回地址，r0 用来传递函数的返回值。此外，在 ARM 中，寄存器是 32 位的，因此正好是 int 型，所以函数的返回值类型和参数类型都是 int 型。既然链接寄存器 lr 中保存了程序的返回地址，那么汇编程序中 mov pc，lr 指令将 lr 中的值加载到 pc 中即可实现子程序的返回。

第 2 行用 extern 声明了一个外部标号 loop。有了前面的讲解，不难理解 loop 没有返回值，也不需要参数，因此是 void loop(void)。

第 5 行调用函数 sum()，并且参数是 2 和 3，则此时 r0=2，r1=3，lr=返回地址，然后程序跳转到汇编语言程序 sum 处开始执行，执行完后返回值保存在 r0 中，此时 r0=5。

第 6 行判断 val 的值是否等于 5，如果等于 5，则跳转到 loop()处执行。

接下来就是启动 AXD 调试器（参见 3.4.4 节“启动 AXD 调试器”）进行仿真调试，并观察实验现象。

启动 AXD 后，程序停止在第 1 条指令处，如图 5-2 所示。

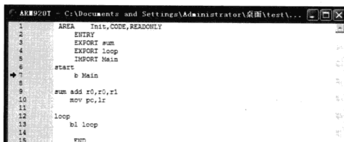


图 5-2 程序开始

单击“step in”按钮（或者按键盘上的 F8 键），程序跳转到 C 程序 Main()处，然后再单击“step in”按钮（或者按键盘上的 F8 键），当程序执行到 val = sum(2,3)时，再单击“step in”按钮，则发现程序跳转到汇编程序 sum 处，并且函数的两个参数被传递到寄存器 r0 和 r1 中，并且可看到，寄存器 r14 中保存了程序的返回地址，如图 5-3 所示。

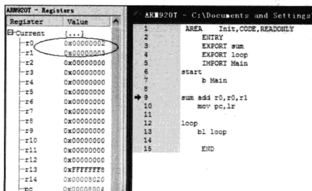


图 5-3 参数传递情况示意图

再次单击“step in”按钮，发现函数的返回值保存在r0中。继续单击“step in”按钮，程序跳转到C程序中接着执行，并且val的值确实是5。

通过这个简单的实验，可以很容易地理解ARM汇编语言和C语言混合编程的基本思路，其中主要涉及的问题有参数如何传递、返回值如何传递、函数调用结束后如何正确返回等。

下面对APCS规则进行简要讲解。读者不需要给予太多的关注，因为在初学阶段不会涉及复杂的混合编程问题，读者只需要有个大概的了解即可，等真正需要的时候再进行深入的学习。

5.2 APCS规则概述

APCS（ARM Process Call Standard）即ARM过程调用规则，定义了一系列的规则来保证ARM汇编语言程序和C语言程序之间能够协调工作，其中涉及函数参数的传递问题、返回值的传递问题以及函数调用过程中寄存器的使用、堆栈的使用等问题。下面对几个常用的规则进行讲解。

5.2.1 寄存器的使用

APCS规定的寄存器的使用规则：R0～R3用来传递函数的参数；R4～R11用来保存函数的局部变量；R13（sp）用做堆栈指针，用来保存当前处理器模式的栈顶指针；链接寄存器R14（lr）用来保存子程序的返回地址。

现在回顾在上面的例子中，函数sum的参数只有两个，因此传递参数时只用到了寄存器R0和R1，程序的返回地址保存在链接寄存器R14(lr)中。

5.2.2 参数传递

当子程序的参数个数小于等于4个数时，参数传递可以通过寄存器R0～R4来实现。当参数个数大于4个时，还需要借助堆栈来传递参数。对于初学者而言，一般不会涉及这种情况，为了降低学习的难度，暂时可以不予考虑。

5.2.3 函数的返回值

如果函数的返回值是个32位的整数，则一般是通过寄存器R0来传递的。如果结果是64位整数，则此时只用寄存器R0传递是无法完成的。在这种情况下，函数的返回值可以通过寄存器R0和R1来传递。

5.3 本章小结

C语言编程灵活方便，便于移植，但是对某些寄存器无法直接访问；汇编语言可以直接控制寄存器，编译效率高，但是编程不方便且不利于移植。因此，在程序开发过程中需

要结合两种语言的优点，各取所长，这就是采取混合编程的原因所在。虽然 ARM 支持汇编语言和 C 语言混合编程，但是在函数调用过程中的参数传递、返回值传递以及寄存器的使用方面需要符合特定的规则（APCS）才能真正混合编程。本章只是简要地进行了阐述，但这些知识足以满足读者在入门阶段的需要。在掌握了基础知识后，参考相关资料即可很快掌握混合编程。

第6章

GPIO 编程实验

前面章节讲述了进行 ARM 开发所需要的开发工具和基础知识，在本章中将向读者展示如何点亮一个 LED，以此来打开硬件开发、程序下载及调试的大门。就像软件开发中的“Hello World”一样，在硬件开发中简单地点亮一个 LED 背后隐藏着许多知识，有硬件方面的，也有软件方面的。只要能顺利点亮 LED，相信其他的硬件操作都是大同小异的事情。可能控制寄存器更复杂一些，也可能控制信号更多一点等，但本质上跟点亮 LED 是很相似的。

通过本章的学习，希望读者能够熟练掌握在 ARM 开发中编程、编译、烧写及程序调试的基本方法和步骤，能够结合 S3C2440 数据手册，掌握 GPIO 硬件的基本构成，进一步理解如何通过程序来控制硬件工作。

6.1 GPIO 概述

GPIO（General Purpose Input/Output）即通用输入/输出端口，本质上就是一些引脚，可以通过程序的控制使这些引脚输出高电平或者低电平。当然，也可以读取这些引脚的电平值。例如，在按键控制中会涉及读取引脚电平的问题。

6.1.1 GPIO 引脚介绍

S3C2440 处理器共有 130 个功能可选择的 I/O 端口，共分为 9 组：GPA、GPB，…，GPJ，其中每组中的 I/O 引脚数目不等。例如，GPB 组共有 11 个 I/O 端口，分别是 GPB0、GPB1、GPB2，…，GPB10；GPC 组有 16 个 I/O 端口，分别是 GPC0、GPC1、GPC2，…，GPC15。所谓功能可选择是指可以通过设置控制寄存器来将某个引脚设为输入、输出或者其他功能。

GPIO 编程是最基本的技能，是其他硬件控制的基础。因此，初学者必须掌握通过程序控制 I/O 的方法。

6.1.2 GPIO 特性分析

前文提到对某个引脚而言，可以通过设置寄存器来将其设为输入、输出或者特殊功能。例如，TQ2440 开发板的 LED1 接在 GPB5 端口上，因此就可以通过设置控制寄存器来将

GPB5 设为输出；此外，只有当 GPB5 引脚输出低电平时，LED1 才会亮。怎么能使 GPB5 输出低电平呢？S3C2440 内部有相应的数据寄存器，数据寄存器的每一位控制一个 I/O 引脚。例如，GPB5 由数据寄存器的第 5 位控制。当向数据寄存器第 5 位写入 0 时，GPB5 便输出低电平；当向数据寄存器第 5 位写入 1 时，GPB5 便输出高电平。

虽然 S3C2440 的 I/O 端口分为 9 组，但是每一组的寄存器都是相似的，只要掌握了一组，其他组都是类似的。

6.1.3 GPIO 相关寄存器

S3C2440 的 GPIO 共分为 9 组，每一组的寄存器都相似。寄存器总体上可以分为三类：控制寄存器 GPxCON、数据寄存器 GPxDAT 和上拉电阻使能寄存器 GPxUP，其中 x 为 A、B，…，J。例如，GPB 的三种寄存器分别是 GPBCON、GPBDAT 和 GPBUP；GPC 的寄存器分别是 GPCCON、GPCDAT 和 GPCUP。

1．控制寄存器 GPxCON

GPxCON 主要用于引脚功能选择。GPB～GPJ 的寄存器操作相同，GPxCON 的每两位控制一根引脚的功能，选择：00 表示设为输入，01 表示设为输出，10 表示特殊功能，11 保留。

例：将 GPB5 设为输出功能。

```
1    #define rGPBCON    (*(volatile unsigned *)0x56000010)
2    rGPBCON &= ~(3 << 10) ;
3    rGPBCON |= (1<<10) ;
```

第 1 行用#define 定义了 rGPBCON，以下对 rGPBCON 的访问就是对控制寄存器 GPBCON 的访问，读者可以参见本书 4.1.3 节。

第 2 行先将 GPBCON 的第 10、11 位清零，这两位用于控制 GPB5，读者可以参见 S3C2440 数据手册。

第 3 行将 GPBCON 的第 10、11 位设为 01，即将 GPB5 设置为输出功能。注意：第 2、3 行对寄存器的访问采用了“先与后或”的方式，目的是不影响寄存器 GPBCON 中其他的位，读者可以参见 4.2.5 节的分析。

需要特别指出的是，对于 GPA 组的 I/O 端口而言，操作方式不一样。虽然 GPA 组共有 25 个 I/O 端口，但是最高两位是保留的，并没有使用，因此真正使用的只有 GPA0～GPA22，共 23 根引脚。当某位被设为 0 时，该引脚被设为输出功能；当某位被设为 1 时，相应的引脚被用做地址线或者产生与地址控制有关的信号，并且此时 GPADAT 将不再起作用。通常情况下，系统外扩存储器时需要将 GPACON 设为全 1，读者有个大概了解即可。

2．数据寄存器 GPxDAT

当引脚被设为输入时，读取 GPxDAT 寄存器即可判断相应引脚的状态。当引脚被设为输出时，写此寄存器的相应位就可令引脚输出高电平或低电平。

3．上拉电阻使能寄存器 GPxUP

当 GpxUP 寄存器的某位为 1 时，与其对应的引脚内部上拉电阻无效；该位为 0 时，与

其对应的引脚内部上拉电阻使能。上拉电阻和下拉电阻如图 6-1 所示。

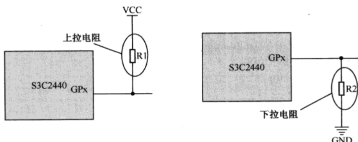


图 6-1 上拉电阻和下拉电阻

使用上拉电阻主要是基于以下几个方面的考虑：对 COMS 芯片而言，引脚悬空容易受到外界的电磁干扰，因此为了防止静电造成对器件造成损坏，不用的引脚不能悬空，一般接上一个电阻，当电阻另一端接电源时称该电阻为上拉电阻，即将引脚电平上拉到高电平，当电阻另一端接地时，称该电阻为下拉电阻，即将引脚电平拉到低电平；芯片的引脚加上拉电阻来提高输出电平，从而提高芯片输入信号的噪声容限，进而增强抗干扰能力。

6.1.4 GPIO 应用实例

访问 GPIO 的操作一般有两种情况：控制引脚输出高、低电平，检测引脚电平。例如，控制引脚输出高、低电平可以实现点亮或熄灭 LED；检测引脚电平可以检测按键是否按下。这两种情况都是通过对特定寄存器的读/写来实现的。

例：TQ2440 开发板上共有 4 个 LED，其接口电路如图 6-2 所示。

现在的问题是：LED 是怎么被点亮的呢？简单点就是，只要 LED 中流过足够的电流，它就发光。现在就是不知道这个足够的电流是多少。其实，在上述 LED 接口电路中，当 GPB5 输出低电平时，电阻和 LED 两端的总电压为 3.3V，那么有初中欧姆定律的知识就可以很容易地判断流过 LED 的电流不会超过 3.3 mA，因为 LED 也有内阻。因此就可以判断，这种 LED 发光时的电流是小于 3.3 mA 的，一般 1～2 mA 就会发光。

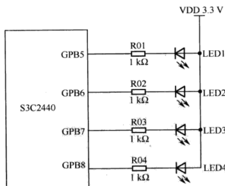


图 6-2 LED 接口电路

现在最关键的问题是：如何用程序控制 GPB5 输出低电平？

基本操作方法是：

（1）写 GPBCON 寄存器，将 GPB5 设为输出功能。

（2）写 GPBUP 寄存器，将 GPB5 上拉电阻使能。

（3）写 GPBDAT 寄存器，将 GPB5 设为输出低电平。

请看下面的程序：

```
1    #define rGPBCON     (*(volatile unsigned *)0x56000010)//Port B control
2    #define rGPBDAT     (*(volatile unsigned *)0x56000014)    //Port B data
3    #define rGPBUP      (*(volatile unsigned *)0x56000018)    //Pull-up control B
4    rGPBCON   &=   ~(3 << 10);
5    rGPBCON   |=   (1<<10) ;
6    rGPBUP    &=   ~ (1 << 5) ;
7    rGPBDAT   &=   ~(1 << 5);
```

第 1 行用#define 定义了 rGPBCON，以下对 rGPBCON 的访问就是对控制寄存器 GPBCON 的访问（详细分析参见 4.1.3 节）。

第 4、5 行先将 GPBCON 第 10、11 位清零，然后写入 01，即可将 GPB5 设为输出功能。注意，这里操作寄存器的方法是，先用按位与指令将相应的位清零，然后用按位或指令将对应的值写入。

第 6 行使 GPB5 上拉电阻使能。

第 7 行向 GPBDAT 寄存器第 5 位写入零（将第 5 位清零即可），进而实现 GPB5 引脚输出低电平。

通过上面的分析，读者只需要了解向寄存器写入数据的方法即可。一般是将按位与运算和按位或运算结合使用，这样可以避免在写入寄存器的过程中破坏其他用不到的位。

6.2 基础实验：第一个裸机程序——流水灯

文字描述的东西，未免使人觉得厌烦，没有什么比自己点亮流水灯更让人觉得兴奋的了！的确，只要点亮了 LED，剩下的工作无非就是不断地学习其他硬件资源的控制方法。

6.2.1 硬件电路分析

前文已经讲述过，TQ2440 开发板上共有 4 个 LED，其接口电路如图 6-3 所示。

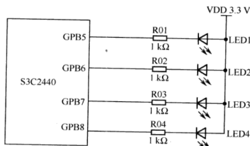


图 6-3　LED 接口电路

LED 发光原理：只要 LED 中流过足够的电流，它就发光。现在就是不知道这个足够的电流是多少。其实，在上述 LED 接口电路中，当 GPB5 输出低电平时，电阻和 LED 两端的总电压为 3.3 V，那么有初中欧姆定律的知识就可以很容易地判断流过 LED 的电流不会超过 3.3 mA，因为 LED 也有内阻。因此，就可以判断，这种 LED 发光时的电流是小于 3.3 mA 的，一般 1～2 mA 就会发光。

6.2.2 建立工程并添加启动代码

关于启动代码，读者可以暂时不予考虑，现在只需要将目标集中在如何点亮一个 LED 上。在第 7 章会着重讨论启动代码都做了哪些事情，因此读者只需要按照下面步骤将启动代码添加到新建立的工程中即可。

（1）参见本书 2.2.1 节的步骤，建立一个新工程（名字叫 ledtest），复制本书光盘 startcode 到 ledtest 目录下，为了以后管理代码方便，请读者在此目录下建立一个文件夹（命名为 source），所有的用户程序都保存在此文件夹下，这样利于源文件的管理，如图 6-4 所示。

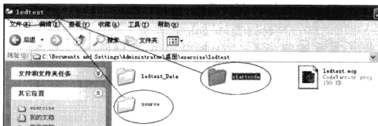


图 6-4　复制 startcode 到 ledtest 目录下

（2）在新工程中单击鼠标右键，选择"Create Group"，如图 6-5 所示。输入名字 startcode，单击"OK"按钮。

（3）右键单击工程窗口中的"startcode"选择"Add Files"，如图 6-6 所示。然后，将在第（1）步中复制到 ledtest 目录的 startcode 文件夹下的除去.inc 后缀的文件外的其他文件全部选中，最后单击"打开"按钮即可，最终效果如图 6-7 所示。

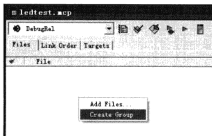


图 6-5　单击鼠标右键选择"Create Group"

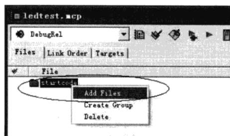


图 6-6　右键单击"startcode"，选择"Add Files"

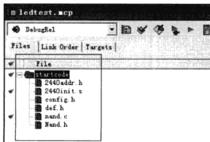


图 6-7　最终效果

6.2.3 添加源文件

提醒读者注意，下面讲述的内容是想给读者展现出模块化编程的全貌，当读者不理解的时候可以按照下面的步骤做，等做完后就会豁然开朗。其实，模块化编程在大型项目开发中的作用还是很明显的。

（1）在刚才建立的工程下建立一个文本文件，以便输入用户程序，单击“New Text File”按钮（如图 6-8 所示），建立三个文件，分别命名为 led.c、led.h、Main.c，最后保存在 ledtest 目录的 source 文件夹下。

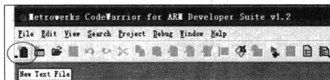


图 6-8　新建文件

（2）在工程窗口中单击鼠标右键，选择“Add Files”（如图 6-9 所示），即可弹出“Select files to add”对话框，选择相应的源文件，单击“打开”按钮即可，如图 6-10 所示。

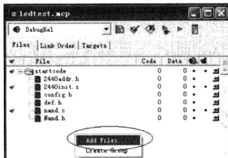


图 6-9　添加文件到工程

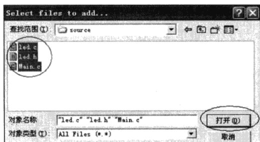


图 6-10　选择源文件

ledtest 源文件总体布局如图 6-11 所示，其中 startcode 文件夹下存放的是启动代码，然后是三个源文件 led.c、led.h 和 Main.c。

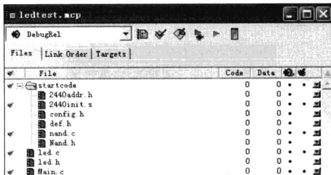


图 6-11　ledtest 源文件总体布局

6.2.4 编辑源文件

本书所有实验例程均采用了模块化编程，因此对模块化编程进行简要的讲解。如果读者对模块化编程没有接触，请大胆地阅读，等再看第二遍时肯定会熟练掌握了。把 led 看成一个模块，该模块包含两个文件：一个是头文件 led.h，里面只包含函数的声明，包含了对 Led_Init()、Led1_On()、Led1_Off()三个函数的声明；另一个是 Led.c 文件，在该文件中对这三个函数进行了实现。在其他文件中就可以只包含 led.h 文件，然后就可以使用 Led_Init()、Led1_On()、Led1_Off()这三个函数了。为了处理重复包含问题还需要一些小技巧，暂且留点悬念吧。在图 6-11 中，双击 led.c 即可打开源文件，接下来输入源文件的内容。

led.c 文件的内容如下：

```
1   #include "led.h"
2   #include "2440addr.h"

3   void Led_Init(void)
    {
4       rGPBCON &= ~(3 << 10);
5       rGPBCON |= (1<<10) ;
6       rGPBUP &=   ~(1 << 5) ;
7       rGPBDAT |=(1 << 5);
    }
8   void Led1_On(void)
    {
9       rGPBDAT &= (~(1 << 5));//LED1ON ;1111 1101 1111
    }
10  void Led1_Off(void)
    {
11      rGPBDAT |= (1 << 5) ;
    }
```

需要注意的是，为了编程方便，把 S3C2440 的所有寄存器定义都放在了 2440addr.h 头文件中，在其他文件中只需要包含此文件即可，如在第 4 行中 rGPBCON 就是在 2440addr.h 中定义的。打开 2440addr.h，读者会发现其定义：#define rGPBCON(*(volatile unsigned *)0x56000010)。关于其他不必过多解释，读者应该可以很容易地看懂上述程序了。如果实在看不懂，请回顾本章 6.1.3 节。唯一需要注意的是第 8 行，当 GPB5 输出低电平时，LED1 亮，因此初始化时应该使 LED1 熄灭，所以需要让 GPB5 输出高电平。

led.h 文件的内容如下：

```
1   #ifndef __LED_H__
2   #define __LED_H__

3   extern void Led_Init(void);
4   extern void Led1_On(void);
```

```
5    extern void Led1_Off(void);
6    #endif
```

第 1、2、6 行是为了解决重复包含的问题。还记得前文提到的重复包含问题吗？读者可以暂不理会，只要记住这个格式即可，毕竟，现在的目标是尽最大努力点亮 LED。

第 3、4、5 行用 extern 关键字声明了三个函数，也就是说在其他文件中可以使用这三个函数。

最后就是 Main.c 文件了，内容如下：

```
1    #include "led.h"
2    int Main()
     {
3        Led_Init() ;
4        while(1)
         {
5            Led1_On() ;
         }
6        return 0;
     }
```

第 1 行包含了 led.h，这样就可以在程序中使用以下三个函数：Led_Init()、Led1_On()、Led1_Off()。

第 3 行调用 Led_Init()完成了对 LED 的初始化。

第 5 行点亮了 LED1，而且无限循环。

下面正确地设置工程。

6.2.5 工程设置、编译、链接

完成上述工作后，需要正确地设置工程。与工程设置、编译、链接有关的按钮如图 6-12 所示。

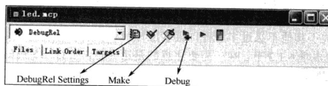


图 6-12　与工程设置、编译、链接有关的按钮

- DebugRel Settings：设置工程的属性，如链接地址的设置、输出文件的格式、编译选项等。
- Make：编译、链接。
- Debug：启动 AXD 调试器。

工程设置的步骤如下。

（1）单击“DebugRel Settings”按钮，在弹出的“DebugRel Settings”对话框左侧选择“Target Settings”，然后在“Post-linker”下拉列表框中选择“ARM fromELF”，如图 6-13 所示。

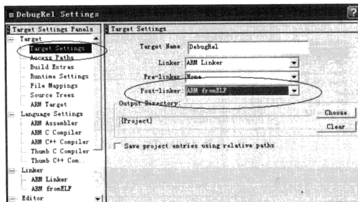


图 6-13　“DebugRel Settings”对话框

（2）在“DebugRel Settings”对话框左侧选择“ARM Assembler”，然后在“Architecture or Processor”下拉列表框中选择“ARM920T”，如图 6-14 所示。

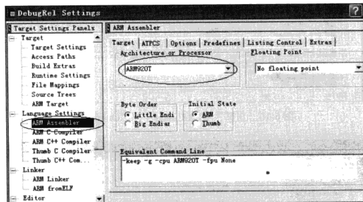


图 6-14　选择“ARM Assembler”

（3）在“DebugRel Settings”对话框左侧选择“ARM C Compiler”，然后在“Architecture or Processor”下拉列表框中选择“ARM920T”，如图 6-15 所示。

（4）在“DebugRel Settings”对话框左侧选择“ARM Linker”，然后在“Output”选项卡的“RO Base”框中输入“0x30000000”，如图 6-16 所示。在“Options”选项卡的“Image entry point”框中输入“0x30000000”，如图 6-17 所示。在 Layout 面板下 Object/Symbol 输入框中输入“2440init.o”，在“Section”框中输入“Init”，如图 6-18 所示，其实这里的 2440init.o 就是由汇编文件 2440init.s 生成的目标文件，Init 是在 2440init.s 文件中定义的一个段，段名叫 Init。

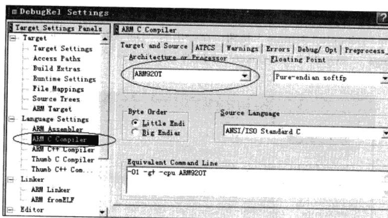


图 6-15　选择“ARM C Compiler”

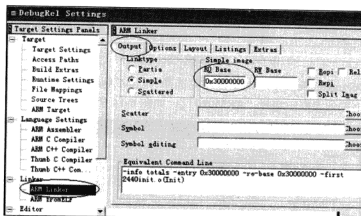


图 6-16　选择“ARM Linker”→“Output”

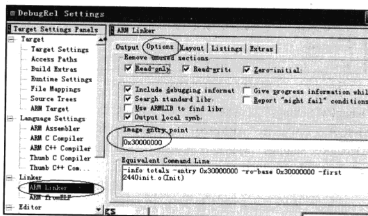


图 6-17　选择“ARM Linker”→“Options”

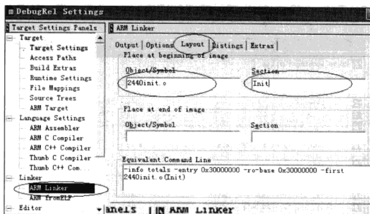


图 6-18 选择“ARM Linker”→“Layout”

（5）在“DebugRel Settings”对话框左侧选择“ARM fromELF”，然后在“Output format”下拉列表框中选择“Plain binary”，在“Output file name”框中输入“ledtest.bin”，如图 6-19 所示，最后单击“OK”按钮即可。

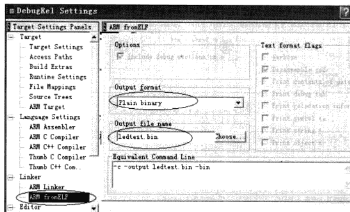


图 6-19 选择“ARM fromELF”

（6）单击图 6-12 中的“Make”按钮，会弹出如图 6-20 所示的对话框。

（7）在 ledtest\ledtest_Data\DebugRel 目录下会看到 ledtest.bin 文件，这就是要下载到 NAND FLASH 中运行的程序代码，如图 6-21 所示。

6.2.6 下载程序到开发板运行

首先讲解笔记本电脑通过 U-Boot 下载程序到 NAND FLASH 的步骤，然后讲解通过 H-JTAG 下载程序到 NAND FLASH 的步骤。

笔记本电脑通过 U-Boot 下载程序到 NAND FLASH 的步骤如下。

（1）选择从 NOR FLASH 启动，打开超级终端，开发板上电后，出现 U-Boot 下载界面，如图 6-22 所示。然后输入字母 a 或者 A。

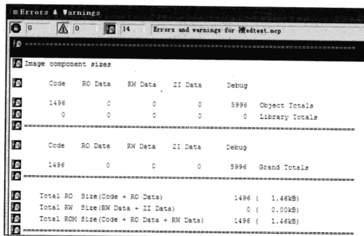


图 6-20 “Errors & Warnings”对话框

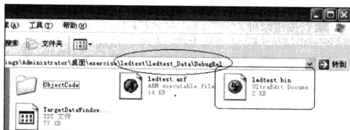
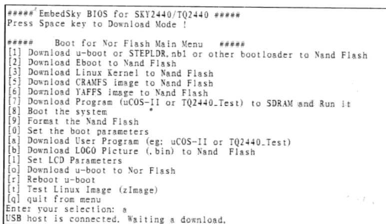


图 6-21 查看 ledtest.bin 文件


```
##### EmbedSky BIOS for SKY2440/TQ2440 #####
Press Space key to Download Mode !

#####    Boot for Nor Flash Main Menu    #####
[1] Download u-boot or STEPLDR.nb1 or other bootloader to Nand Flash
[2] Download Eboot to Nand Flash
[3] Download Linux Kernel to Nand Flash
[5] Download CRAMFS image to Nand Flash
[6] Download YAFFS image to Nand Flash
[7] Download Program (uCOS-II or TQ2440_Test) to SDRAM and Run it
[8] Boot the system
[9] Format the Nand Flash
[0] Set the boot parameters
[a] Download User Program (eg: uCOS-II or TQ2440_Test)
[b] Download LOGO Picture (.bin) to Nand  Flash
[l] Set LCD Parameters
[o] Download u-boot to Nor Flash
[r] Reboot u-boot
[t] Test Linux Image (zImage)
[q] quit from menu
Enter your selection: a
USB host is connected. Waiting a download.
```

图 6-22 U-Boot 下载界面

（2）打开 DNW 软件（关于 DNW 软件的设置，请读者参见《TQ2440 开发板使用手册》），选择“USB Port”，然后选择“Transmit”即可，如图 6-23 所示。然后选择刚才生成的 ledtest.bin 文件即可下载到 NAND FLASH。

图 6-23　USB 下载界面

到此为止，程序的下载工作结束。接下来，选择从 NAND FLASH 启动，然后给开发板上电，第一个 LED 已经被点亮了，兴奋吧？这就是程序开发的全过程。

如果读者使用的是台式机，通过 H-JTAG 下载程序到 NAND FLASH 的步骤如下。

（1）打开 H-JTAG，然后设置，选择“Settings”菜单的“LPT JTAG Settings”项，按如图 6-24 所示进行设置。

图 6-24　H-JTAG 设置

（2）选择“Operation”菜单的“Detect Target”项，H-JTAG 会自动检测 CPU（开发板要上电）。如果检测到，会显示如图 6-25 所示的界面，然后在“Flasher”菜单下选择“Start H-Flasher”项。

图 6-25　选择“Start H-Flasher”

（3）在弹出的 H-Flasher 窗口中，选择“Load”，然后选择“TQ2440_nand_2KP.hfc”即可，如图 6-26 所示。

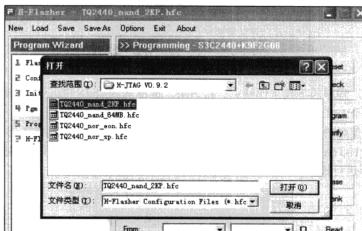


图 6-26 选择配置文件

说明：H-Flasher 配置文件共有 4 个，具体选择哪一个配置文件，请参考具体的开发板硬件配置。

- TQ2440_nand_2KP.hfc：用于烧写 2KB 大页面 Nand Flash 的 H-Flash 配置文件。
- TQ2440_nand_64MB.hfc：用于烧写 512B 页面 Nand Flash 的 H-Flash 配置文件。
- TQ2440_nor_eon.hfc：用于烧写 Nor Flash 的 H-Flash 配置文件。
- TQ2440_nor_sp.hfc：用于烧写 Nor Flash 的 H-Flash 配置文件。

（4）在 H-Flasher 窗口中，选择“Flash Selection”，然后选择“S3C2440+K9F2G08”，如图 6-27 所示。

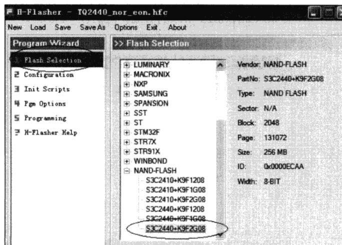


图 6-27 选择 Flash 型号界面

（5）在 H-Flasher 窗口中，选择“Programming”，单击“Check”按钮，会显示出 Flash 型号，单击“Src File”右端的“...”按钮，会弹出“打开”对话框，然后找到“ledtest.bin”，最后单击“Program”按钮即可实现程序下载，整体过程如图 6-28 所示。

（6）程序下载结束后，会显示如图 6-29 所示的界面。

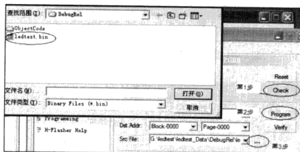


图 6-28　程序下载过程

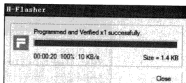


图 6-29　下载完成

程序下载结束了。下面只需要选择从 NAND FLASH 启动，上电后就会发现第一个 LED 已经悄悄地亮起来了。当然，这只是 ARM 开发的第一步。

6.2.7　由点亮 LED 引发的思考

经过不懈的努力，LED 点亮了，那么下面这些问题又该如何解释呢？

（1）到底程序是从哪里执行的呢，第一条指令在哪里呢？

（2）系统上电后是如何一步一步地运行到 Main 函数的呢？

（3）一般编写 C 语言函数都用 main 作为主函数名。这里为什么主函数名用 Main 呢？为什么不用 main 呢？什么情况下才可以用 main 做主函数名呢？

（4）在前文中，新建工程后把启动代码加到了 startcode 中，什么是启动代码呢？启动代码有什么作用呢？

（5）什么叫从 NAND FLASH 启动呢？什么叫从 NOR FLASH 启动呢？

（6）NAND FLASH 是什么？NOR FLASH 又是什么？二者有什么区别？

相信读者可能会有更多的问题。不错，正是这些问题困扰着初学者，他们在论坛发帖求助，搜索资料，但是最终还是有些问题搞不懂……

基于对上述问题的解决，构成本书的编写原始动力。

给初学者的一点建议：要敢于提出问题，并且要将问题记录下来，只要问题提出来了，总会有解决问题的方法。但是如果不把问题提出来，这个问题可能会一直困扰着你，甚至影响你学习 ARM 的积极性。在本书的其他章节中，编者会尽自己的最大努力向读者展现出解决上述问题的思路和方法，必要的时候会涉及部分经典书籍，同样也会推荐给读者。

下面回想一下以前讲过的 AXD 调试器：AXD 调试器不正是调试程序用的吗？可以用它来观察程序执行的全过程吗？可以！对，就是用 AXD 调试器来观察程序的执行过程。当用户程序编写完成后，就可以启动 AXD 调试器进行程序的调试，AXD 调试器支持单步、全速、执行到光标处、断点等调试功能，可以观察变量、寄存器和内存单元的内容。

启动 AXD 调试器有两种方法。

- 单击“开始”→“程序”→“ARM Developer Suite v1.2”→“AXD Debugger”，如图 6-30 所示。
- 在 CordWarrior for ARM 里单击“Debug（调试）”按钮即可启动 AXD 调试器，如图 6-31 所示。



图 6-30　启动 AXD 调试器

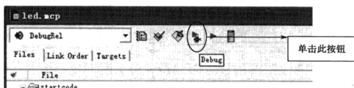


图 6-31　启动 AXD 调试器

这里需要注意的问题是：用第二种方法启动 AXD 调试器时，会自动加载映像文件，但有时会出现不正常的现象。当读者遇到这种现象时，可以尝试用第一种方法启动 AXD 调试器。本书推荐用第一种方法启动 AXD 调试器，但是这时需要手动加载映像文件。

此外，启动 AXD 调试器后还要设置 AXD 调试器，然后加载可执行镜像文件。这些内容请读者参见第 2 章 2.3 节“工程的调试”和第 3 章 3.4 节“用 AXD 调试 ARM 汇编程序实验”。

选择“File”菜单的“Load Image”项，如图 6-32 所示，即可打开“Load Image”对话框，选择相应的映像文件即可，然后选择 ledtest.axf 文件（还记得 ledtest.axf 文件的位置吗？参见图 6-21 可知，在 ledtest\ledtest_Data\DebugRel 目录下），如图 6-33 所示。



图 6-32　装载映像文件

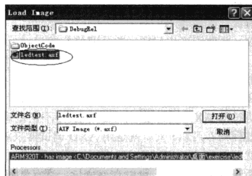


图 6-33　选择.axf 格式的映像文件

最后会出现如图 6-34 所示的界面，可以看到程序是从 b ResetHandler 开始执行的。

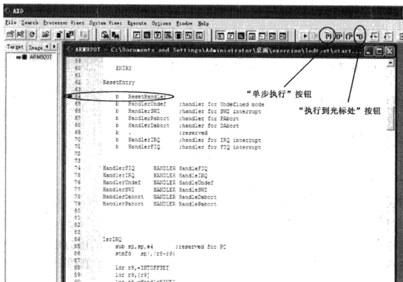


图 6-34　第一条指令开始执行

接下来单击“单步执行”按钮，一步一步地观察程序的执行效果，最终跳到 Main 函数。这里介绍一个小技巧：当遇到一个循环时，可以将光标定位到跳过循环的地方，然后单击图 6-34 中“执行到光标处”按钮即可。通过上述过程的讲解，初学者只需要知道 ARM 程序从 b ResetHandler 开始执行，然后最终执行到 Main 函数处。

在这里，初学者可能对大段的汇编代码感到无从下手，没有关系，正是这些汇编代码构成了传说中的启动代码，在第 7 章会专门讲解启动代码。

在基于 ARM 处理器的嵌入式系统开发中，应用程序大多采用 C 或者 C++等高级语言编写，在运行应用程序之前，需要对系统进行初始化。因此在系统上电后，需要有一段引导程序完成对系统资源的初始化，为用户程序建立基本的运行环境。因此，启动代码主要是完成对 ARM 处理器的初始化，使其能够正常工作。为了给读者留下一个印象，下面给读者展示启动代码到底做了哪些事情：

- 建立异常中断向量表。
- 初始化各模式的堆栈。
- 初始化硬件。
- 最后跳转到主应用程序。
- 看到很多书上都这么写，那么具体是怎么实现的呢？第 7 章将会揭开启动代码的神秘面纱。

6.2.8　再议点亮 LED 实验

再回顾一下点亮 LED 的例子，从下面的 led.c 文件中的代码看到，每个函数里面只有一句，调用函数需要保存函数的返回地址，然后从函数返回时需要将返回地址赋值给 PC，

这些都会使程序执行速度变慢。为了改善这种情况，对于这种程序代码量很小的程序段，可以用宏的形式实现。

```
1    void Led1_On(void)
     {
2        rGPBDAT &= (~(1 << 5));//LED1ON ;1111 1101 1111
     }
3    void Led1_Off(void)
     {
4        rGPBDAT |= (1 << 5) ;
     }
```

在 led.h 中用宏实现上述代码，代码如下：

```
#define Led1_On()        {rGPBDAT &= (~(1 << 5));}
#define Led1_Off()       {rGPBDAT |= (1 << 5);}
```

如果这样定义，则在程序中调用 Led1_On()时，会在调用的地方展开（调用宏，就是在调用的地方展开），可以节省函数调用的开销。图 6-35 展示了修改后整个 ledtest 工程的总体布局。

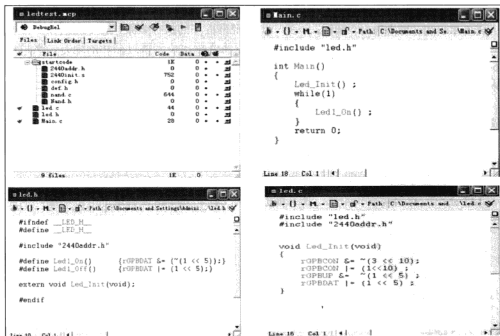


图 6-35　整个 ledtest 工程的总体布局

什么时候用函数，什么时候用宏来实现函数功能呢？读者可以这么理解：当函数就几行代码时，可以用宏来实现函数的功能；当代码量较大时，就用函数来实现，因为如果函

数比较大，假设用宏实现，每次调用该宏都要展开，这样会使程序的代码量加大。用宏实现会加大代码量，用函数实现会增加函数调用的开销，各有利弊，因此需要对两个方面权衡考虑。

6.2.9 将点亮一个LED扩展到流水灯[①]

有了前面的讲解，设计一个流水灯将变得很容易。流水灯基本流程是：点亮 LED1，延时一会儿后再点亮 LED2，延时一会儿后再点亮 LED3，再延时一会儿后点亮 LED4。由于采用模块化程序设计，应当尽量使模块保持独立性，因此延时函数放在一个单独的模块 common.h 和 common.c 中。

common.h 文件的内容如下：

```
1   #ifndef __COMMON_H__
2   #define __COMMON_H__

3   extern void Delay(void);

4   #endif
```

第 1、2、4 行是为了避免多个文件包含 common.h 文件时产生重复包含问题。

第 3 行声明了一个外部函数 Delay()；当其他源文件包含 common.h 时，就可以直接调用 Delay() 。

common.c 文件的内容如下：

```
1   #include "common.h"
3   void Delay(void)
    {
3       int i;
4       for (i=0;i<1000000;i++);
5   }
```

第 1 行包含 common.h，这是模块化编程的基本方法，.h 文件只负责声明函数，.c 文件负责函数的实现，但是需要将.h 文件包含。

第 2～4 行是对 common.h 文件中声明的延时函数 Delay()的具体实现。

对 ledflow.h 文件的内容进行了如下扩展：

```
#ifndef __LEDFLOW_H__
#define __LEDFLOW_H__
#include "2440addr.h"
#define Led1_On()        {rGPBDAT &= (~(1 << 5));}
#define Led1_Off()       {rGPBDAT |= (1 << 5);}
#define Led2_On()        {rGPBDAT &= (~(1 << 6));}
#define Led2_Off()       {rGPBDAT |= (1 << 6);}
```

① 流水灯例程参见本书光盘 exercise 目录下 ledflow，所有文件全在 ledflow 目录下。

```
#define Led3_On()        {rGPBDAT &= (~(1 << 7));}
#define Led3_Off()       {rGPBDAT |= (1 << 7);}
#define Led4_On()        {rGPBDAT &= (~(1 << 8));}
#define Led4_Off()       {rGPBDAT |= (1 << 8);}
extern void Led_Init(void);

#endif
```

可见，只是对其他三个 LED 进行了扩展，操作方法跟操作 LED1 是一样的道理。此外，ledflow.h 文件最后声明了一个外部函数 Led_Init()。经过前面的讲解，可以推测该函数一定是在 ledflow.c 文件中进行的实现。

下面是 ledflow.c 文件的内容：

```
#include "ledflow.h"
#include "2440addr.h"

void Led_Init(void)
{
    rGPBCON &= ~((3 << 10) | (3 << 12) | (3 << 14) |(3 << 16));
    rGPBCON |= ((1<<10) | (1<<12) | (1<<14) | (1<<16)) ;
    rGPBUP   &= ~((1 << 5) | (1 << 6) | (1 << 7) || (1 << 8) ) ;
    rGPBDAT |= (1 << 5) | (1 << 6) | (1 << 7) | (1 << 8) ;
}
```

可见，对寄存器的操作主要是按位与操作和按位或操作。按位与操作主要用于对某几位的清零，按位或操作主要用于对某几位的置 1。尤其需要注意的是最后一行，使 LED1～LED4 所对应的引脚输出高电平。为什么呢？因为当输出低电平时，相应的 LED 被点亮，所以初始化时要让其熄灭。

Main.c 文件中没有定义新函数，只是对各个模块所实现函数的调用，内容如下：

```
#include "ledflow.h"
#include "common.h"
int Main()
{
    Led_Init() ;
    while(1)
    {
        Led1_On() ;Delay() ;Led1_Off() ;
        Led2_On() ;Delay() ;Led2_Off() ;
        Led3_On() ;Delay() ;Led3_Off() ;
        Led4_On() ;Delay() ;Led4_Off() ;
    }
    return 0;
}
```

然后，按照 6.2.4～6.2.6 节所讲解的步骤，将.bin 格式的文件下载到 NAND FLASH 中，然后从 NAND FLASH 启动，会发现 LED1～LED4 在轮流点亮，即实现了流水灯。

6.3 GPIO 扩展实验

上一节讲解了点亮 LED 的基本步骤，对模块化编程进行了简要的分析，同时给出了 led 模块中 led.h 和 led.c 实现过程，下面将对其进行必要的补充和扩展了几个实验。

6.3.1 按键实验

到此为止，LED 实验就结束了。下面介绍按键实验，道理都是一样的。

1．硬件电路分析

按键接口电路如图 6-36 所示。以 GPF1 为例，GPF1 通过一个 10 kΩ上拉电阻接到电源，因此当按键没有按下时，GPF1 引脚电平为高电平；当按键按下时，引脚电平会变为低电平。因此，程序中就是通过对 GPF1 引脚的电平进行不断的检测，当引脚电平为低电平时说明按键被按下。

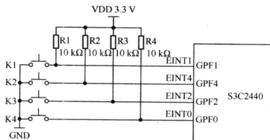


图 6-36 按键接口电路

2．建立工程并添加源文件

首先建立工程，然后编辑源文件，设置工程，编译生成可执行文件。读者可以按照前面讲解的步骤做。这里介绍一个简单的方法：复制 ledtest 文件夹，重命名为 keytest，然后把刚才重命名的文件夹 keytest 下的 ledtest.mcp 重命名为 keytest.mcp，最后把 ledtest_Data 文件夹删除，最终效果如图 6-37 所示。最后把 source 文件夹下的 led.h、led.c 两个文件复制并重命名为 key.h、key.c。这种方法的好处是，不需要设置工程（因为 ledtest.mcp 中已经设置好了，参见 6.2.4 节），编辑完源文件，直接单击“Make”按钮即可。

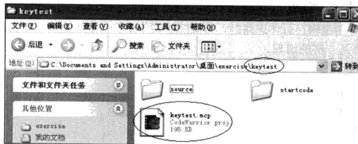


图 6-37 工程文件图

此时，如果直接双击“keytest.mcp”会出现如图 6-38 所示的错误提示，这是软件自身的原因所导致的。

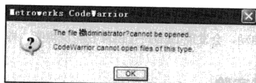


图 6-38　ADS 打开工程时常见错误

解决这个问题有两种方法，如图 6-39 所示。

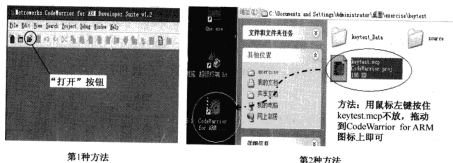


图 6-39　两种打开工程的方法

- 单击"OK"按钮，然后在 ADS 中单击"打开"按钮，找到 keytest.mcp 即可打开。
- 直接将 keytest.mcp 拖动到 ADS 快捷方式图标上面即可打开。

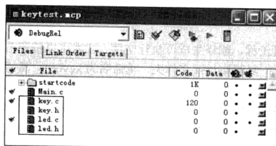


图 6-40　keytest 工程文件结构

打开 keytest.mcp 后，将 key.h、key.c 加到工程中（添加文件到工程的方法见 6.2.3 节），最后，keytest 工程文件结构如图 6-40 所示。

3．编辑源文件

由于采用了模块化编程，本工程共两个模块：一个模块是 led，另一个模块是 key。对于 led 模块 led.h、led.c 文件，已经在前文中讲述过，因此下面主要是对 key.h、key.c 文件的编辑。总体思路是，当 KEY1 按下时，点亮 LED1。只要这个功能实现，其他按键的操作是一样的道理。

下面是 key.h 文件的内容：

```
1   #ifndef __KEY_H__
2   #define __KEY_H__

3   #define KEY1 (0 << 2)        //GPF1 接 key1

4   extern void Key_Init() ;
5   extern int Key_Scan() ;
```

```
6    #endif
```

有了前面的分析，相信读者能顺利地看懂 key.h 文件。

下面是 key.c 文件的内容：

```
1    #include "key.h"
2    #include "2440addr.h"
3    void Key_Init(void)
     {
4        rGPFCON &= ~(3 << 0) ;
5        rGPFCON |= KEY1 ;
6        rGPFDAT |= (1 << 1) ;//将 KEY1 对应的引脚 GPF1 输出为高电平
     }
7    int Key_Scan(void)
     {
8        int keynum = 0 ;
9        if((rGPFDAT & (1 << 1)) == 0 )
         {
10           keynum = 1 ;
         }
11       return keynum ;
     }
```

第 4、5 两行实现将 KEY1 对应的引脚 GPF1 配置成输入。

第 6 行使 GPF1 输出高电平。

第 9 行判断 GPF 数据寄存器 GPFDAT 的第 1 位是否为 0，即判断 GPF1 引脚是否是低电平。若为低电平，则 keynum 赋值为 1，否则返回为 0。

4．下载到开发板运行

首先需要将输出的可执行二进制文件名改为 keytest.bin，如图 6-41 所示。

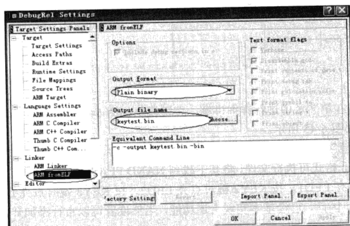


图 6-41　设置要生成的可执行文件格式

然后单击“Make”按钮即可，最后在\keytest\keytest_Data\DebugRel 目录下会看到 keytest.bin 文件，如图 6-42 所示，按 6.2.6 节讲述的步骤将其下载到 NAND FLASH 即可。

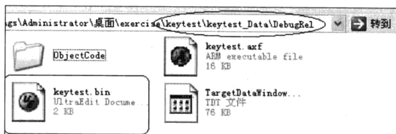


图 6-42 .bin 格式可执行文件的位置

选择从 NAND FLASH 启动，按 KEY1 键，发现 LED1 被点亮，说明实验结果与前面的分析吻合。

5. 按键实验扩展

经过前面的讲解，读者应该很容易将其他三个按键进行扩展，实现功能：按 KEY1 键时点亮 LED1，按 KEY2 键时点亮 LED2，按 KEY3 键时点亮 LED3，按 KEY4 键时点亮 LED4。文件总体布局如图 6-43 所示。

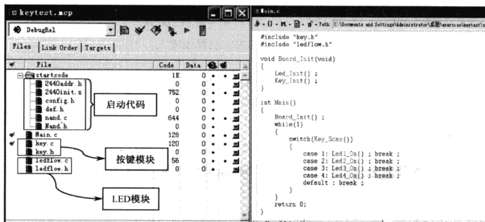


图 6-43 文件总体布局

启动代码将在第 7 章中进行详细讲解，按键模块如图 6-44 所示。

LED 模块如图 6-45 所示。

只需要按照前文讲述的步骤，建立工程，编辑源文件，设置工程，然后单击“Make”按钮即可生成.bin 格式可执行文件，将其下载到 NAND FLASH 中，从 NAND FLASH 启动，按 KEY1 键时点亮 LED1，按 KEY2 键时点亮 LED2，按 KEY3 键时点亮 LED3，按 KEY4 键时点亮 LED4。因此，已经基本实现上述功能，当然读者可以自行添加其他功能，在此不做赘述。



图 6-44　按键模块

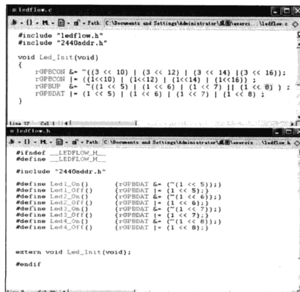


图 6-45　LED 模块

6.3.2 蜂鸣器实验

有了前面的讲解，模块化编程的基本思路已经逐步展现给读者，下面通过一个蜂鸣器的实验练习一下。在本实验中，对于工程的建立、工程的设置、Make 以及程序下载到 NAND FLASH 中等步骤不做赘述，读者可以参考前面的步骤。下面重点讲解蜂鸣器驱动原理以及蜂鸣器模块是如何实现的。

1. 硬件电路分析

蜂鸣器接口电路如图 6-46 所示，由原理图知道 TOUT0 与 GPB0 相连接，因此当 GPB0 输出高电平时，三极管导通，蜂鸣器蜂鸣；当 GPB0 输出低电平时，三极管截止，蜂鸣器停止蜂鸣。三极管 8050 起到了驱动蜂鸣器的作用，即将 GPB0 输出的电流放大到能使蜂鸣器蜂鸣的电流值。

VDD_5V J4B 6 6 5 5 CON6_2.0 BUZZER 1 2 BUZZER TOUT0 R801 100 Ω R802 5.1 kΩ VT8 8050 GND

图 6-46 蜂鸣器接口电路

三极管 8050 的放大倍数典型值取 β_{BE}（数据手册是 h_{BE}）$=100$，最大集电极电流 $I_{CM}=1\ 200\ \text{mA}$，R801 用于限制基极电流，则基极电流由如下公式计算：

$$I_b=\frac{3.3-U_{be}}{100}-\frac{U_{be}}{5100}=\frac{3.3-0.7}{100}-\frac{0.7}{5100}=25.9\ \text{mA}$$

假设此时三极管工作在放大状态，则集电极电流 $I_{CM}=\beta\cdot I_b=100\cdot 29.5=2\ 590\ \text{mA}$，当通过蜂鸣器的电流超过 40 mA 时，蜂鸣器上面的压降就会达到 5V，此时，$U_{ec}<U_{eb}$，则三极管工作在深度饱和导通状态，为蜂鸣器提供足够的电流。

2. 蜂鸣器模块程序分析

beep.h 文件的内容如下：

```
1    #ifndef __BEEP_H__
2    #define __BEEP_H__
3    #include "2440addr.h"

4    #define Beep_On()       {rGPBDAT |= (1 << 0);}
5    #define Beep_Off()      {rGPBDAT &= ~(1 << 0);}

6    extern void Beep_Init(void);
7    #endif
```

第 4、5 行通过宏定义实现了对蜂鸣器的开关操作。当 GPB0 输出高电平时，蜂鸣器蜂鸣；当 GPB0 输出低电平时，蜂鸣器停止蜂鸣。

第 6 行声明了一个外部函数，该函数是在 beep.c 中实现的。

beep.c 文件的内容如下：

```
1    #include "beep.h"
2    #include "2440addr.h"
```

```
3     void Beep_Init(void)
      {
4         rGPBCON &= ~(3 << 0) ;
5         rGPBCON |= (1<<0) ;
6         rGPBUP   &=   ~(1 << 0) ;
7         rGPBDAT &= ~(1 << 0);
      }
```

分析：beep.c 文件对函数 Beep_Init 函数进行了初始化。

第 4、5 行将 GPB0 设置为输出口。

第 6 行设置上拉电阻功能。

第 7 行使 GPB0 输出低电平，关闭蜂鸣器。

有了上面的模块定义，则在 Main.c 文件中只需要调用相应的函数即可实现蜂鸣器的蜂鸣。

```
1     #include "beep.h"
2     #include "common.h"
3     int Main()
      {
4         Beep_Init() ;
5         while(1)
          {
6             Beep_On() ; Delay() ;
7             Beep_Off() ; Delay() ;
          }
8         return 0;
      }
```

第 2 行包含了 common.h 文件，参见 6.2.9 节，在 common 模块中只定义了一个延时函数。

第 4 行调用了蜂鸣器初始化函数 Beep_Init()。

第 6、7 行实现了蜂鸣器的蜂鸣。

beep.mcp 工程的总体布局如图 6-47 所示。

最后设置工程，由 Make 生成.bin 格式的文件，下载到 NAND FLASH 中，启动即可听到蜂鸣器间断地蜂鸣。

6.4 本章小结

本章通过几个实例对 ARM 裸机开发进行了具体讲解，可能有的地方显得比较烦琐，但考虑到初学者面临的各种问题，还是将具体细节展现出来了。当然，在上面这些例子中，唯一的缺点可能是，程序代码功能太单一，不过本章主要是想给读者展现出模块化开发的全貌，将代码功能放在次要的位置。如果读者可以很好地掌握本章提供的技巧和方法，相信在本书后面的学习中会逐渐体会到模块化开发带来的优越性。



图 6-47 beep.mcp 工程的总体布局

6.5 扩展阅读之模块化编程、NAND FLASH 和 NOR FLASH 概述

刚刚接触 ARM 的初学者在学习过程中会遇到各种各样的问题，其中模块化编程、NAND FLASH 和 NOR FLASH 的区别就是其中之一，下面对其进行简要的讲解。

- 模块化编程分析与设计

通过对前面几个实验的分析，读者可能对模块化编程有了初步的了解。在理想的模块化编程中，各个模块可以看做是一个个的黑盒子，只需要注意模块提供的功能，不需要关心具体实现该功能的策略和方法，即提供的是机制而不是策略，机制即功能，策略即方法。好比用户买了一部 iPhone，只需要会用它所提供的各种功能即可，至于各种功能是如何实现的，用户不需要关心。

在大型程序开发中，一个程序由不同的模块组成，可能不同的模块会由不同的人员负责。在编写某个模块的时候，很可能就需要调用别人写好的模块的接口。这个时候关心的是：其他模块提供了什么样的接口，应该如何去调用，至于模块内部是如何实现的，对于调用者而言，无须过多关注。模块对外提供的只是接口，把不需要的细节尽可能对外部屏蔽起来，正是采用模块化程序设计所需要注意的地方。

一个模块包含两个文件：一个是.h 文件（又称为头文件），另一个是.c 文件。

.h 文件可以理解为一份接口描述文件，其文件内部一般不包含任何实质性的函数代码，可以把这个头文件理解成为一份说明书，其内容就是这个模块对外提供的接口函数或接口变量。此外，该文件也可以包含一些很重要的宏定义（如前文中 led.h 中实现的宏 Led1_On()）

以及一些数据结构的信息，离开了这些信息，该模块提供的接口函数或接口变量很可能就无法正常使用。头文件的基本构成原则是：不该让外界知道的信息就不应该出现在头文件里，而供外界调用的模块内接口函数或接口变量所必需的信息就一定要出现在头文件里，否则，外界就无法正确地调用该模块提供的功能。当外部函数或者文件调用该模块提供的接口函数或变量时，就必须包含该模块提供的这个接口描述文件—.h 文件（头文件）。同时，该模块的.c 文件也需要包含这个模块头文件（因为它包含了模块源文件中所需要的宏定义或数据结构等信息）。通常，头文件的名字应该与源文件的名字保持一致，这样便可以清晰地知道哪个头文件是对哪个源文件的描述。

.c 文件主要功能是对.h 文件中声明的外部函数进行具体的实现，对具体实现方式没有特殊规定，只要能实现其函数的功能即可。

下面再回顾一下 ledtest 工程文件的总体布局（如图 6-48 所示）。



图 6-48　ledtest 工程文件的总体布局

该工程包含一个 led 模块，其中分为 led.h 和 led.c，led.h 只包含供外部调用宏定义 Led1_On()、Led1_Off()以及函数 Led_Init()，led.c 文件对函数 Led_Init()进行了实现，同时包含了头文件 led.h。Main.c 文件中要调用 Led_Init()和 Led1_On()，因此包含了头文件 led.h。

- NAND FLASH 和 NOR FLASH 概述

初学者刚接触 TQ2440 开发板时，可以说对 NAND FLASH 和 NOR FLASH 没有什么概念。简单地讲，NAND FLASH 相当于计算机的硬盘，适合存储大量数据；NOR FLASH 相当于计算机的内存，但是数据掉电后不丢失。因此，从 NAND FLASH 启动就相当于从硬盘启动，这里涉及 S3C2440 内部的 Stepping Stone 的支持，在 NAND FLASH 实验部分会

详细讲解。

NAND 和 NOR 是现在市场上两种主要的非易失闪存技术。1988 年，Intel 公司首先开发出 NOR FLASH 技术，彻底改变了原先由 EPROM 和 EEPROM 独占的局面。1989 年，东芝公司发布了 NAND FLASH 结构，强调降低每比特的成本，并且实现了像磁盘一样可以通过接口轻松升级。

NOR FLASH 的特点是芯片内执行（eXecute In Place，XIP），这样应用程序可以直接在 NOR FLASH 闪存内运行，不必再把代码读到系统 RAM 中。NAND FLASH 提供极高的单元密度，可以达到高存储密度，并且写入和擦除的速度也很快。应用 NAND FLASH 的难点在于对 NAND FLASH 的操作需要特殊的控制器来产生所需要的时序。

第 2 篇
提　高　篇

第 7 章 启动代码分析

在前面章节中，对 ARM 裸机开发基础知识和程序的编译、下载及调试进行了详细的讲解，但是对启动代码并没有涉及。在本章中，将对启动代码进行着重讲解。因为学习 ARM 裸机开发需要对系统的引导有一定的了解，此外通过对启动代码的讲解，为以后学习操作系统移植打下一个好基础。

启动代码主要是负责对板级硬件的初始化，以及程序代码的搬移等。编写启动代码需要对处理器有一定的了解，对初学者而言，这无疑是一个巨大的挑战，但是读者只需要根据本章所讲述的内容，理解启动代码是具体怎么实现的即可。本章力图详细，争取不遗漏任何一个细节，目的是尽可能给初学者展现出一个启动代码的具体实现过程。

在讲解启动代码之前，需要对 TQ2440 开发板的硬件配置情况有个清楚的认识。在第 1 章中只是展示了 NAND FLASH、NOR FLASH、SDRAM 在核心板的布局。在本章中需要进一步了解上述器件的型号以及在系统启动阶段的工作流程。

7.1 从开发板硬件讲起

编写启动代码还需要从具体的硬件配置讲起，那么具体需要哪些硬件配置呢？总结起来主要有中断控制器、内部时钟电路、存储器控制器，一般还需要了解 SDRAM 的一些参数，此外还需要对 NAND FLASH 及 NAND FLASH 控制器有所了解。很多初学者面临的一个问题是：从 NAND FALSH 启动的具体流程是怎么样的？本章将详细讨论这些问题。

关于 SDRAM 和 NAND FLASH 的基本操作，读者可以暂时不必关心，在本书后面章节中由相关的实验具体讲述。在本章中，读者只需要深入理解启动代码做了哪些事情，按照怎样的顺序做这些事情即可。

7.1.1 TQ2440 核心板芯片功能介绍

TQ2440 开发板上的 NAND FLASH、NOR FLASH 及 SDRAM 如图 7-1、图 7-2 所示。核心板上各器件的型号和容量如表 7-1 所示。

注意：不同时间生产的 TQ2440 开发板所使用的器件型号可能略有不同。因此，初学者应该看清具体的型号。

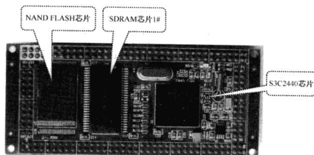


图 7-1 核心板正面俯视图

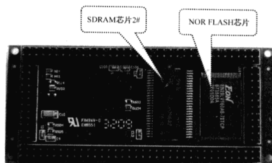


图 7-2 核心板反面俯视图

表 7-1 核心板上各器件的型号和容量

芯片名称	型号	容量
NAND FLASH	K9F2G08U0B	256 MB
NOR FLASH	EN29LV160AB	2 MB
SDRAM	MT48LC16M16A2	32 MB

这些芯片的主要作用是什么？开发板上电后，程序是如何启动的呢？下面将详细讲解。

- NAND FLASH 就相当于计算机的硬盘，容量较大，一般用于存储数据，但是无法对其进行直接寻址，也就是说，对NAND FLASH的访问需要有专门的NAND FLASH控制器来产生相应的时序。
- NOR FLASH 容量较小，但是 CPU 可以对其进行直接寻址，因此可以在 NOR FLASH 中执行程序。
- SDRAM 就相当于计算机的内存，主要用于执行程序，但是掉电后里面的数据会丢失。NOR FLASH 中的数据掉电后不会丢失，这是 SDRAM 与 NOR FLASH 的主要区别。

S3C2440 处理器支持两种启动方式：从 NAND FLASH 启动和从 NOR FLASH 启动，启动方式可以通过开发板上的 NAND/NOR FLASH 启动选择开关进行选择。需要说明的是，虽然程序无法从 NAND FLASH 执行，但是 S3C2440 内部有一个 “Stepping Stone” 的缓冲区，其大小是 4 KB，上电后会通过硬件逻辑自动地复制 NAND FLASH 前 4KB 的数据到 “Stepping Stone” 中，然后在 “Stepping Stone” 中开始执行。

S3C2440 处理器支持 4 种启动方式（如表 7-2 所示），具体采用哪种启动方式由系统本身的硬件情况决定，表 7-2 中 OM1 和 OM0 是 S3C2440 处理器的两个引脚，可以通过开关来改变引脚的电平值，进而实现不同的启动方式选择。因此，具体启动流程为：上电后检测 OM1 和 OM0 的引脚电平，根据不同的电平值来选择具体的启动方式。以 TQ2440 开发板为例，在 BANK0 处接了 16 位的 NOR FLASH 芯片，因此如果从 NOR FLASH 启动，启动时需要使 OM[1:0]=01；如果从 NAND FLASH 启动，需要使 OM[1:0]=00。

表 7-2　启动方式

OM1(Operating Mode 1)	OM0(Operating Mode 0)	启动时 ROM 的数据位宽
0	0	NAND FLASH
0	1	16-bit
1	0	32-bit
1	1	测试模式（Test Mode）

7.1.2　从 NAND FLASH 和 NOR FLASH 启动流程分析

NAND FLASH 与 S3C2440 处理器的接口电路如图 7-3 所示，NAND FLASH 接在了 S3C2440 处理器的内部 NAND FLASH 控制器上，NAND FLASH 控制器能够产生访问 NAND FLASH 所需的时序。

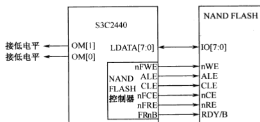


图 7-3　NAND FLASH 与 S3C2440 处理器的接口电路

系统从 NAND FLASH 启动流程如图 7-4 所示。

启动流程分析：如图 7-3 所示，系统上电后，有硬件自动复制 NAND FLASH 中前 4KB 的数据到“Stepping Stone”中，然后在“Stepping Stone”中执行，一般在这 4 KB 的代码中会实现将 NAND FLASH 中的所有应用程序搬移到 SDRAM 中（此时需要借助对 NAND FLASH 进行读操作的函数实现），然后跳到 SDRAM 中接着执行。

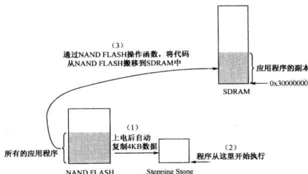


图 7-4　NAND FLASH 启动流程图

NOR FLASH 与 S3C2440 处理器的接口电路如图 7-5 所示，可以观察到 NOR FLASH 的数据总线位宽是 16 位（D[15:0]）因此当从 NOR FLASH 启动时需要使 OM[1:0]=01。由此可以推知：如果在具体设计中所使用的 NOR FLASH 芯片的数据总线宽度是 32 位，从 NOR FLASH 启动时，需要使 OM[1:0]=10。

CPU 可以对 NOR FLASH 进行直接寻址，即可以直接执行 NOR FLASH 中的程序，因从 NOR FLASH 启动时，程序会从第一条指令处开始执行，当然在 NOR FLAHS 中的执行速度比在 SDRAM 中的执行速度慢。通常的做法是，上电后执行一小段程序，实现将程序从 NOR FLASH 中搬移到 SDRAM 中，然后跳转到 SDRAM 中接着执行。

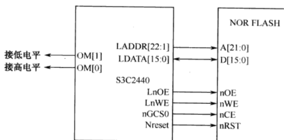


图 7-5 NOR FLASH 与 S3C2440 处理器的接口电路

思考：为什么会有两种启动方式呢？

这是由两种 FLASH 的不同特点决定的。

NAND FLASH 容量大，存储单位比特数据的成本低，但是需要按照特定的时序对 NAND FLASH 进行读/写操作，因此 CPU 无法对 NAND FLASH 中的数据进行直接寻址，CPU 对 NAND FLASH 中数据的读/写是通过专门的 NAND FLAHS 控制器来进行的，因此 NAND FLASH 更适合于存储数据。

NOR FLASH 容量小，速度快，对 NOR FLASH 进行读/写时输入地址，然后给出读/写信号即可从数据总线上得到数据，但是价格一般比 NAND FLASH 高，因此适合做程序存储器。

因此，NOR FLASH 可以直接连接到 ARM 的总线上，但是 NAND FLASH 需要通过 NAND FLASH 控制器与 S3C2440 相连接。

7.2 启动代码详解

总体上来讲，裸机开发所需要的启动代码只有 100 多条汇编指令。通过启动代码的学习，读者可以进一步熟悉 ARM 汇编指令，为以后熟练运用 ARM C 语言、汇编语言混合编程以及 U-Boot 的移植打下一个良好的基础。很多读者有这样的疑问：学习 ARM 汇编到什么程度才算是过关呢？其实只要能看懂启动代码即可。当然，初学阶段并不需要完完整整地把启动代码写出来，只要能看懂现有的启动代码，然后结合自己的理解稍做修改即可。对编写启动代码没有很严格的要求，读者完全可以在学习好一个版本的启动代码后，根据自己的理解，选择自己熟悉的 ARM 指令来实现启动代码。

注意：初学者可以略过这一部分，因为启动代码中不仅涉及了较多的汇编指令，而且结合指令的特点进行了恰当的取舍，尽量精简指令条数。

推荐的学习方法：直接学习本书后面的裸机开发实验，遇到不明白的地方可以参考本章的讲解，在做过一定数量的实验后就能很容易地理解启动代码的功能。

下面结合启动代码进行具体分析：

```
1         GET option.inc
2         GET memcfg.inc
3         GET 2440addr.inc
```

第 1～3 行用 GET 伪操作包含了三个文件。注意，在汇编语言文件中被包含的文件扩展名为.inc。其中，option.inc 文件主要包含了栈地址、中断服务程序基地址以及与系统时钟初始化相关的几个常量定义；memcfg.inc 文件主要包含了与 SDRAM 初始化相关的几个参数定义；2440addr.inc 文件主要包含了 S3C2440 处理器特殊功能寄存器地址的定义。完全可以将这些文件中的常量定义写在一个文件中，但是这样不便于查找和修改，应分开写在 3 个单独的文件中以便于修改和查找。

注意：这三条语句不能顶格书写，否则编译器报错。

```
4   USERMODE      EQU    0x10
5   FIQMODE       EQU    0x11
6   IRQMODE       EQU    0x12
7   SVCMODE       EQU    0x13
8   ABORTMODE     EQU    0x17
9   UNDEFMODE     EQU    0x1b
10  MODEMASK      EQU    0x1f
11  NOINT         EQU    0xc0
```

第 4～11 行主要是将当前程序状态寄存器 CPSR 中模式控制位进行了定义，只是为了编程方便，也可以不定义上述常量，直接给 CPSR 中 M0～M4 赋值也可以实现处理器工作模式的切换。

需要注意的是，用 EQU 定义常量时，需要顶格书写，否则编译器报错。

```
12  UserStack    EQU   (_STACK_BASEADDRESS-0x3800)   ;0x33ff4800 ~
13  SVCStack     EQU   (_STACK_BASEADDRESS-0x2800)   ;0x33ff5800 ~
14  UndefStack   EQU   (_STACK_BASEADDRESS-0x2400)   ;0x33ff5c00 ~
15  AbortStack   EQU   (_STACK_BASEADDRESS-0x2000)   ;0x33ff6000 ~
16  IRQStack     EQU   (_STACK_BASEADDRESS-0x1000)   ;0x33ff7000 ~
17  FIQStack     EQU   (_STACK_BASEADDRESS-0x0)      ;0x33ff8000 ~
```

第 12～17 行定义了各个模式堆栈的起始地址，这样只需要将其初始地址赋给相应模式的堆栈指针即可实现各个模式堆栈的初始化。堆栈一般分配在什么地方呢？理论上讲，只要是没有使用的连续内存片都可以当做堆栈用，但是考虑到堆栈设置不当时可能会产生溢出，因此一般将堆栈放在内存的高地址处。

注意：用 EQU 定义常量时，需要顶格书写，否则编译器报错。分号表示注释的开始，分号后面的内容为注释。

```
18       MACRO
19  $HandlerLabel    HANDLER $HandleAddr
20  $HandlerLabel
21       sub     sp, sp, #4
22       stmfd   sp!, {r0}
```

```
23      ldr      r0, =$HandleAddr
24      ldr      r0, [r0]
25      str      r0, [sp, #4]
26      ldmfd    sp!, {r0, pc}
27      MEND
```

该宏实现的功能是：将$HandleAddr 表示的地址赋值给程序计数器 PC，进而实现程序跳转到$HandleAddr 指向的函数处去执行。

第 18～27 行定义了一个宏（定义宏是为了编程时“偷懒”，将很多地方都需要的代码定义成一个宏，在需要引用的地方调用宏即可），在程序中可以通过如下方式调用该宏：

```
HandlerIRQ  HANDLER  HandleIRQ
```

宏展开的结果如下：

```
HandlerIRQ
(1)     sub  sp,  sp, #4
(2)     stmfd     sp!, {r0}
(3)     ldr       r0, = HandleIRQ
(4)     ldr       r0, [r0]
(5)     str       r0, [sp, #4]
(6)     ldmfd     sp!, {r0, pc}
```

由宏展开的结果可以看出，HandlerIRQ 代替了宏定义中的标号$HandlerLabel，同时传递给宏的参数 HandleIRQ 代替了宏定义中的形式参数$ HandleAddr。

上述程序执行过程分析如下。

第（1）～（2）行将堆栈指针减 4，然后将 r0 寄存器的值入栈（因为下面的程序段需要用到寄存器 r0，为了防止 r0 中的数据被破坏，因此需要将其入栈），执行完这两条指令后，堆栈中的数据分布如图 7-6 所示。注意，ARM 处理器规定的堆栈是满递减堆栈。即入栈时先将堆栈指针减 4，然后数据入栈，堆栈指针总是指向刚刚入栈的数据，因此第（1）行指令将堆栈指针减 4 后，执行第（2）行 r0 入栈时，首先是 sp 再减 4，然后 r0 中的数据入栈，所以才会出现如图 7-6 所示的情况。

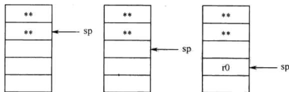


图 7-6 堆栈中的数据分布

第（3）～（4）行将内存单元 HandleIRQ（从后面第 197 行可以看到，在内存中分配的一个 4 字节的存储单元，该内存单元的标号是 HandleIRQ，在汇编中，标号代表一个地址值）中的数据加载到寄存器 r0 中，在 HandleIRQ 内存单元中存储的是中断服务函数的地址，因此第（4）行实际是将中断服务函数的地址加载到了寄存器 r0 中。

第（5）行将 r0 中的数据（中断服务函数的地址）存入到 sp+4 地址单元中。注意，此时堆栈指针并没有变化，执行到此处，堆栈中的数据分布如图 7-7 所示，ISR 代表中断服务

函数的地址。

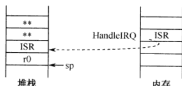


图 7-7　中断服务函数存储到 sp+4 地址处

第（6）行出栈操作，寄存器 r0 中的数据恢复，同时将中断服务函数的地址赋给了程序计数器 PC，因此中断服务函数得以执行。

深入理解： 该宏的功能是将$HandleAddr 表示的地址赋值给程序计数器 PC，进而实现程序跳转到$HandleAddr 指向的函数处去执行。结合上述分析可知，宏展开后的功能是将中断处理函数的地址赋值给程序计数器 PC，进而实现程序跳转到中断处理函数处去执行。

```
28        IMPORT   |Image$$RO$$Base|       ; Base of ROM code
29        IMPORT   |Image$$RO$$Limit|      ; End of ROM code (=start of ROM data)
30        IMPORT   |Image$$RW$$Base|       ; Base of RAM to initialise
31        IMPORT   |Image$$ZI$$Base|       ; Base and limit of area
32        IMPORT   |Image$$ZI$$Limit|      ; to zero initialise
33        IMPORT   Main                    ; The main entry of mon program
34        IMPORT   RdNF2SDRAM              ; Copy Image from Nand Flash to SDRAM
```

第 28～32 行用 IMPORT 伪操作告诉编译器该文件中要引用 Image$$RO$$Base 等几个外部符号，这几个标号是由编译器自动生成的用于标记各个段（ARM 可执行映像文件由 RO 段、RW 段和 ZI 段组成）的起始地址，如表 7-3 所示。

表 7-3　编译器生成的符号

编译器生成的符号	含　义
|Image$$RO$$Base|	RO 段起始地址
|Image$$RO$$Limit|	RO 段结束地址+1
|Image$$RW$$Base|	RW 段起始地址
|Image$$RW$$Limit|	RW 段结束地址+1
|Image$$ZI$$Base|	ZI 段起始地址
|Image$$ZI$$Limit|	ZI 段结束地址+1

第 33～34 两行，因为在该汇编文件中需要用到 Main 函数和 RdNF2SDRAM 函数，但是这两个函数是在其他文件中定义的，因此也需要使用 IMPORT 伪操作，IMPORT 伪操作的作用类似于 C 语言中的 extern 关键字。

```
35        AREA   Init, CODE, READONLY
36        ENTRY
37    ResetEntry
38        b      ResetHandler
39        b      HandlerUndef        ;handler for Undefined mode
40        b      HandlerSWI          ;handler for SWI interrupt
41        b      HandlerPabort       ;handler for PAbort
42        b      HandlerDabort       ;handler for DAbort
```

```
43        b       .                    ;reserved
44        b       HandlerIRQ           ;handler for IRQ interrupt
45        b       HandlerFIQ           ;handler for FIQ interrupt
```

第 35 行才是启动代码的真正开始，用 AREA 伪操作定义了一个段（Init）。

第 38～45 行是几条跳转指令，这就是异常向量表，如图 7-4 所示，每个跳转指令对应着一种类型异常处理。

当某种类型的异常发生时，ARM 处理器便强制把 PC 指针指向异常中断向量表中对应的异常向量处，然后从向量表跳转到存储器里存放异常中断服务程序的地址，执行具体的异常中断服务程序。例如上电时，处理器会把 PC 指针强制指向 0 地址处，然后执行跳转指令 b ResetHandler；当软中断发生时，处理器会把 PC 指针强制指向 0x08 地址处，然后执行跳转指令 b HandlerSWI。

0x0000001c	b HandlerFIQ
0x00000018	b HandlerIRQ
0x00000014	b .
0x00000010	b HandlerDabort
0x0000000C	b HandlerPabort
0x00000008	b HandlerSWI
0x00000004	b HandlerUndef
0x00000000	b ResetHandler

图 7-8 异常向量表

```
46    HandlerFIQ         HANDLER HandleFIQ
47    HandlerIRQ         HANDLER HandleIRQ
48    HandlerUndef       HANDLER HandleUndef
49    HandlerSWI         HANDLER HandleSWI
50    HandlerDabort      HANDLER HandleDabort
51    HandlerPabort      HANDLER HandlePabort
```

ARM 要求异常向量表必须存放在从 0 地址开始的连续 8×4 字节的地址空间内，如图 7-8 所示，因为每种中断只占据向量表中 1 个字的存储空间。因此，一般情况下异常向量表中 0x00-0x1c 这段程序只是简单地包含跳转指令，程序从此处跳转到具体的中断处理函数处去执行。

经过前面的分析知，该宏实现的功能是将相应的异常处理函数的地址赋值给程序计数器 PC，程序跳转到异常处理函数处去执行。第 46～51 行进行了宏展开，也就是将各种异常处理函数的地址赋值给程序计数器 PC，实现对异常的处理。

在上述异常处理中，一般情况会用到 3 种：上电复位、外部中断和软中断。本书第 11 章向读者展示 ARM 处理器软中断的机制。

```
52    IsrIRQ
53        sub      sp, sp, #4           ;reserved for PC
54        stmfd    sp!, {r8-r9}
55        ldr      r9, =INTOFFSET
56        ldr      r9, [r9]
57        ldr      r8, =HandleEINT0
58        add      r8, r8, r9, lsl #2
59        ldr      r8, [r8]
60        str      r8, [sp, #8]
61        ldmfd    sp!, {r8-r9, pc}
```

```
62      LTORG
```

注意： 第 52～62 行程序主要涉及 ARM 对中断的处理，内容稍微复杂一点，需要读者对 ARM 的中断系统有一定的了解。初学者可以暂时跳过此处，当阅读到第 11 章时再学习这部分内容也可以。

上述程序段的功能：进行第二次查表，找到中断服务函数的地址，然后将该地址赋值给程序计数器 PC，进而执行相应的中断服务函数。

S3C2440 处理器外部中断有 EINT0、INT_ADC 等共 32 个中断源。在用户可以通过控制中断模式寄存器 INTMODE 来定义某个中断是属于 IRQ 模式的情况下，当这 32 个中断源中某一个发出中断请求时，CPU 都跳转到 b HandlerIRQ。为了能确定到底是哪一个中断发生，需要在 RAM 区开辟一段内存空间，建立一张软件设定的中断向量表（参见后面第 190～230 行），用来存放各个中断服务程序的入口地址，每个中断服务程序的入口地址都占用一个表项，1 个字的空间。在应用程序中将中断服务程序入口地址写入相应的表项内，完成中断向量的设置。

对于上述 32 个外部中断，当某一个外部中断发生时，中断偏移寄存器 INOFFSET 会被硬件置为相应的值。32 个外部中断源与中断偏移寄存器 INOFFSET 的值的对应关系如表 7-4 所示。

表 7-4　32 个外部中断源与中断偏移寄存器 INOFFSET 的值的对应关系

中 断 源	INOFFSET 值	中 断 源	INOFFSET 值
INT_ADC	31	INT_UART2	15
INT_RTC	30	INT_TIMER4	14
INT_SPI1	29	INT_TIMER3	13
INT_UART0	28	INT_TIMER2	12
INT_IIC	27	INT_TIMER1	11
INT_USBH	26	INT_TIMER0	10
INT_USBD	25	INT_WDT_AC97	9
INT_NFCON	24	INT_TICK	8
INT_UART1	23	nBATT_FLT	7
INT_SPI0	22	INT_CAM	6
INT_SDI	21	EINT8_23	5
INT_DMA3	20	EINT4_7	4
INT_DMA2	19	EINT3	3
INT_DMA1	18	EINT2	2
INT_DMA0	17	EINT1	1
INT_LCD	16	EINT0	0

例如：当外部中断 0 发生时，INOFFSET 会被硬件置为 0；当定时器 0 中断发生时，INOFFSET 会被硬件置为 10。

第 53 行首先使堆栈指针减 4，这是为保存具体中断服务函数地址预留的空间。

第 54 行将 r8、r9 的值入栈保护，因为下面要用到这两个寄存器。此时，堆栈的数据分

布情况如图 7-9 所示。

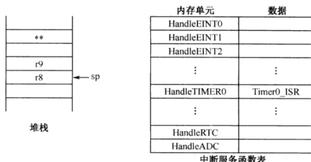


图 7-9 第 53、54 两行程序执行后，堆栈的数据分布情况

第 55、56 两行取出中断偏移寄存器 INOFFSET 的值，将其加载到寄存器 r9 中。

第 57 行将中断服务函数表的首地址加载到寄存器 r8 中，即此时 r8 指向了中断服务函数表的首地址。

第 58 行将 r9 中的数据为偏移量查找上述中断服务函数表。例如，当定时器 0 中断发生时，中断偏移寄存器 INOFFSET 的值会被硬件置为 10，因为中断服务函数表中每一项占 4 个字节，因此 r9 中的数据左移 2 位(左移 2 位相当于乘以 4)，即为 HandleTIMER0 距离中断服务函数表的偏移量，在该地址处存放了定时器 0 的中断服务函数地址 Timer0_ISR。

第 59 行将定时器 0 的中断服务函数地址 Timer0_ISR 加载到寄存器 r8 中(还是以定时器 0 中断为例讲解)。

第 60 行将定时器 0 中断服务函数的首地址存储到 sp+8 处，如图 7-10 所示。

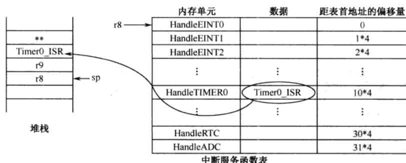


图 7-10 将定时器 0 中断服务函数的首地址加载到 sp+8 处

第 61 行出栈，此时定时器 0 中断服务函数的地址会被加载到程序计数器 PC 中，进而执行定时器 0 的中断服务函数。

第 62 行用 LTORG 声明了一个文字池，一般用 ldr 伪指令将 32 位立即数加载到寄存器中时需要声明一个文字池。

扩展：尽管还没有对 ARM 的中断处理过程进行讲解，但是为了更形象地展示 ARM 处理器对外部中断的响应过程，读者可以结合图 7-11 加以理解。

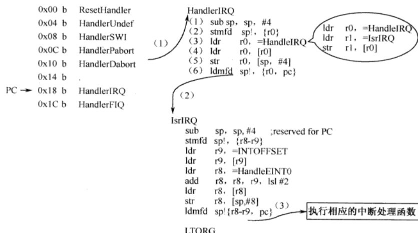


图 7-11 ARM 处理器对外部中断的响应过程

注意：简单地讲，ARM 处理器响应外部中断时需要两次跳转才能找到具体的中断服务函数。如图 7-11 所示，当外部中断发生时，程序的执行流程如下。

（1）硬件会强制性地将程序计数器 PC 指向中断向量表的 0x18 处，一般在此处放置一条跳转指令。

（2）执行 HnadlerIRQ 程序段，在此程序段中取得 IsrIRQ 的地址，进而跳转到 IsrIRQ 处进行第 2 次查表，找到具体的中断服务函数地址。

（3）跳转到具体的中断服务程序处去执行，进而实现对某种类型外部中断的响应。

（4）执行中断返回。

对于上述流程，读者可以结合中断部分的实验，进而深入地理解，在此可以大概浏览一下。

```
63  ResetHandler
64      ldr  r0, =WTCON          ;watch dog disable
65      ldr  r1, =0x0
66      str  r1, [r0]
```

对于 S3C2440 处理器，系统上电后是从 0 地址处开始执行的，在此地址处是一条跳转指令 b ResetHandler，因此上电后会执行该跳转指令即可跳到 64 行开始执行（注意，ResetHandler 是个标号，因此顶格书写，同时该标号指示了第 64 行指令的地址）。

第 64 行将看门狗控制寄存器（WTCON 是在 2440addr.inc 文件中定义的）的地址加载到寄存器 r0 中，此处 ldr 是一条伪指令，加载 32 位的立即数到寄存器中。

第 65、66 两行将 0 写入看门狗控制寄存器。关闭看门狗的方法：向看门狗控制寄存器写 0 即可。因此，这段程序实际上是关闭看门狗。系统初始化阶段需要关闭看门狗。

```
67      ldr  r0, =INTMSK
68      ldr  r1, =0xffffffff     ;all interrupt disable
69      str  r1, [r0]
70      ldr  r0, =INTSUBMSK
```

```
71        ldr   r1, =0x7fff            ;all sub interrupt disable
72        str   r1, [r0]
```

在系统初始化阶段需要将中断关闭，对于 S3C2440 处理器只需要向中断屏蔽寄存器 INTMSK 和子中断屏蔽寄存器 INTSUBMSK 写入全 1 即可。

```
73        ldr   r0, =LOCKTIME
74        ldr   r1, =0xffffff
75        str   r1, [r0]

76        ldr   r0, =CLKDIVN
77        ldr   r1, =CLKDIV_VAL
78        str   r1, [r0]

79        [ CLKDIV_VAL>1             ; means Fclk:Hclk is not 1:1.
80        mrc   p15, 0, r0, c1, c0, 0
81        orr   r0, r0, #0xc0000000
82        mcr   p15, 0, r0, c1, c0, 0
83        |
84        mrc p15, 0, r0, c1, c0, 0
85        bic r0, r0, #0xc0000000
86        mcr p15, 0, r0, c1, c0, 0
87        ]
    ;Configure UPLL
88        ldr   r0, =UPLLCON
    ;Fin = 12.0MHz,   UCLK = 48MHz
89        ldr   r1, =((U_MDIV<<12)+(U_PDIV<<4)+U_SDIV)
90        str   r1, [r0]
91        nop        ; at least 7-clocks delay must be inserted for setting hardware be completed.
92        nop
93        nop
94        nop
95        nop
96        nop
97        nop
    ;Configure MPLL
98        ldr   r0, =MPLLCON
    ;Fin = 12.0MHz,   FCLK = 200MHz
99        ldr   r1, =((M_MDIV<<12)+(M_PDIV<<4)+M_SDIV)
100       str   r1, [r0]
```

第 73～100 行是系统时钟初始化部分，也是系统初始化阶段需要特别注意的地方

第 73～75 行是向寄存器 LOCKTIME 写入相应的数值，这个值对应着图 7-13 中的 Lock Time，第 74 行的 0xffffff 是 S3C2440 数据手册上给出的默认值，一般按照这个值初始化

LOCKTIME 寄存器即可满足要求。

第 76～78 行向寄存器 CLKDIVN 中写入相应的值 CLKDIV_VAL（CLKDIV_VAL 是在 option.inc 文件中定义的），该值决定了 FCLK、HCLK 和 PCLK 的分频比。在本书所用启动代码中将 CLKDIV_VAL 设定为 3，则 FCLK:HCLK:PCLK=1:2:4。

注意：查询 S3C2440 数据手册可知，CLKDIV_VAL 的值与 FCLK、HCLK 和 PCLK 之间的分频比关系如下：

- 当 CLKDIV_VAL=0 时，FCLK:HCLK:PCLK = 1:1:1。
- 当 CLKDIV_VAL=1 时，FCLK:HCLK:PCLK = 1:1:2。
- 当 CLKDIV_VAL=2 时，FCLK:HCLK:PCLK = 1:2:2。
- 当 CLKDIV_VAL=3 时，FCLK:HCLK:PCLK = 1:2:4。
- 当 CLKDIV_VAL=4 时，FCLK:HCLK:PCLK = 1:4:4。
- 当 CLKDIV_VAL=5 时，FCLK:HCLK:PCLK = 1:4:8。
- 当 CLKDIV_VAL=6 时，FCLK:HCLK:PCLK = 1:3:3。
- 当 CLKDIV_VAL=7 时，FCLK:HCLK:PCLK = 1:3:6。

第 79～87 行是一个选择结构，“[” 相当于 if，“|” 相当于 else，“]” 相当于 endif。参考 S3C2440 数据手册知道，当 FCLK 不等于 HCLK 时，也就是当 CLKDIV_VAL 大于 1 时需要将处理器的工作模式从快速模式切换到异步模式。第 80～82 行就是切换处理器工作模式的指令，这涉及对几个协处理器的访问，初学者可以不必关心，在 S3C2440 数据手册上给出了上述代码，编写启动代码时参考即可。

第 88～97 行是初始化 USB 时钟，本书所有实验均没有涉及，因此不再详细讲解。

第 98～100 行是向寄存器 MPLLCON 中写入数值，前面提到 MPLLCON 寄存器控制 FCLK 和 Fin 之间的比例关系，具体的计算将在第 8 章详细讲解，读者在此只需要了解经过上述初始化后 CPU 核的工作频率 FCLK 已经产生，并且 FCLK、HCLK 和 PCLK 之间的比例关系也已经确定了，即完成了处理器时钟的初始化。

```
        ;Set memory control registers
101         adrl        r0, SMRDATA         ;please caution!
102         ldmia       r0, {r1-r13}
103         ldr         r0, =BWSCON
104         stmia       r0, {r1-r13}
```

第 101～104 行实现对存储器控制器的初始化，其实就是对 13 个寄存器的初始化。

第 101 行用 adrl 伪指令加载数据表的首地址到寄存器 r0。

第 102 行用批量加载指令 ldmia 将 r0 指向的数据表中的 13 个参数（每个参数占 4 个字节）加载到寄存器组 r1～r13 中。

第 103 行用 ldr 伪指令将 S3C2440 处理器存储器控制器的起始地址加载到寄存器 r0 中。

第 104 行用批量存储指令 stmia 将寄存器组 r1～r13 中的数据依次存储到 r0 指向的 13 个存储器控制寄存器中。

图 7-12 向读者展现了初始化 SDRAM 过程详解。

注意：第 101 行将数据表的首地址 SMRDATA 加载到寄存器 r0 中，采用的是 adrl 伪

指令，并没有采用 ldr 伪指令（用 ldr 伪指令的格式为：ldr r0,=SMRDATA），这里涉及位置无关代码（PIC）的问题，初学者可以暂时记住这里的用法，关于位置无关代码的详细解释，请读者参考《深入理解计算机系统结构》（Randal E. Bryant 著，龚奕利、雷迎春译），该书第 7 章“链接”对位置无关代码（PIC）进行了详细的分析。

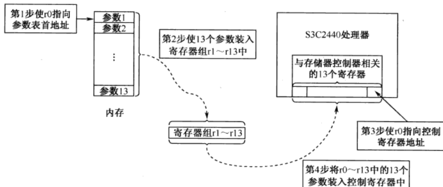


图 7-12　初始化 SDRAM 过程详解

```
105       bl      InitStacks
```

第 105 行是跳转到标号 InitStacks 处执行堆栈初始化。注意：这里用的跳转指令是 bl，即执行完堆栈初始化后返回到该指令的下一条指令处接着执行。

```
;*****************************************************************
    106       ldr        r0,  =BWSCON
    107       ldr        r0,  [r0]
    108       ands       r0,  r0,  #6          ;OM[1:0] != 0,   NOR FLash boot
    109       bne        copy_proc_beg          ;do not read nand flash
    110       adr        r0,  ResetEntry        ;OM[1:0] == 0,   NAND FLash boot
    111       cmp        r0,  #0                ;if use Multi-ice
    112       bne        copy_proc_beg          ;do not read nand flash for boot
```

第 106～112 行才是选择从 NAND FLASH 或者 NOR FLASH 启动的关键。

第 106～107 行将 BWSCON 寄存器里的数据读入寄存器 r0。从 S3C2440 数据手册上可以查到 BWSCON 寄存器第 1、2 两位，反映了系统总线宽度的信息。

第 108 行，and 指令主要用于测试数据的某几位是 0 还是 1，后面的 s 表示该指令的运算结果影响标志位。因此，该条指令主要是测试 r0 第 1、2 两位。当运算结果是 0 时，说明 r0 的第 1、2 两位是 0，即 BWSCON 寄存器的第 1、2 两位是 0，即系统是从 NAND FLASH 启动的，则执行第 110 行程序；当运算结果不为 0 时，说明是从 NOR FLASH 启动，再执行第 109 行程序。

第 109 行，执行该条指令的前提是系统从 NOR FLASH 启动，然后跳转到 copy_proc_beg 标号处执行，其功能是：实现代码从加载域到运行域的搬移工作，这也是启动代码的核心。

第 110～111 行，比较难理解，这时可以借助反汇编以后的代码理解。如图 7-13 所示

可见，adr　r0，ResetEntry 指令用 SUB r0，pc，#0x1a0 代替了，而由于执行到该指令时程序计数器 PC 实际上是指向了下两条指令的地址，即 PC = 0x1a0，所以，执行完该指令后，r0 的值为 0。

```
0x00000188:   e3a00448   H...   MOV   r0,#0x48000000
0x0000018c:   e5900000   ....   LDR   r0,[r0,#0]
0x00000190:   e2100006   ....   ANDS  r0,r0,#6
0x00000194:   1affffffe  ....   BNE   copy_proc_beg   ; 0x1ac
0x00000198:   e24f0f68   h.O.   SUB   r0,pc,#0x1a0 ; #0
0x0000019c:   e3500000   ..P.   CMP   r0,#0
0x000001a0:   1afffffe   ....   BNE   copy_proc_beg   ; 0x1ac
```

图 7-13　NAND FLASH 启动反汇编代码

小技巧：adr 伪指令是将标号基于 PC 的相对地址加载到寄存器中，因为是将程序下载到了 NAND FLASH 的 0 地址处，所以 ResetEntry 的地址为 0。

第 112 行，由上面的分析知道，该行指令不会执行。

```
;***********************************************************************
113  nand_boot_beg
114       bl    RdNF2SDRAM
115       ldr   pc,  =copy_proc_beg
;***********************************************************************
```

第 113 行，只是一个程序标号，没有实际的用处，可以忽略。

第 114 行，调用 C 语言函数 RdNF2SDRAM，将代码从 NAND FLASH 读入到内存中。

第 115 行，用 ldr 指令将 copy_proc_beg 的绝对地址加载到程序计数器 PC 中，即跳转到了内存中区执行。

到此为止，从 NAND FLASH 启动的流程已经完全讲清楚了。下面结合图 7-14 详细分析从 NAND FLASH 启动总流程。

（1）系统上电后，S3C2440 处理器通过硬件电路自动将 NAND FLASH 中的前 4KB 代码复制到内部的 Stepping Stone 中，该 Stepping Stone 就是一块 RAM，可以执行程序。

（2）程序从 Stepping Stone 的 0 地址处开始执行第 1 条指令。

（3）程序执行到 bl　RdNF2SDRAM（第 114 行）时，调用 NAND FLASH 函数，将 NAND FLASH 中的代码全部复制到 SDRAM 中。

（4）执行 ldr pc，=copy_proc_beg，将 copy_proc_beg 的绝对地址加载到程序计数器 PC 中。

（5）程序跳转到 SDRAM 中 copy_proc_beg 标号处去执行。

知识扩展：第（4）步中提到了 copy_proc_beg 的绝对地址，那么什么是绝对地址呢？绝对地址是程序编译链接后确定的，如图 7-15 所示，在链接时，指定的 entry 地址是 0x30000000，这个 entry 即第 36 行的 ENTRY，也就是说，在最终编译链接后生成的可执行程序中，copy_proc_beg 的绝对地址=0x30000000+偏移量。这个偏移量是多少呢？就是 copy_proc_beg 距离 ENTRY 的偏移量。因此，在第（4）步执行完后，虽然 Stepping Stone 中也有 copy_proc_beg，SDRAM 中也有 copy_proc_beg，但是程序并没有跳到 Stepping Stone 中的 copy_proc_beg 处执行，而是跳到了 SDRAM 中的 copy_proc_beg 处去执行。

图 7-14 从 NAND FLASH 启动总流程

图 7-15 设置 entry 地址

ZI(未初始化的数据段)
RW(可读/写的数据段)
RO(代码和只读数据段)

图 7-16 ARM 映像文件结构（加载域）

一个简单的可执行程序的映像文件结构如图 7-16 所示。它由 RO、RW 和 ZI 三个段组成，其中 RO 为代码和只读数据段；RW 为可读 / 写的数据段；ZI 为未初始化的数据段。

此外，映像文件还可以分为加载域（Load View）

和运行域（Execution view）。加载域反映了 ARM 可执行映像文件各个段存放在存储器中时的位置关系，运行域反映了 ARM 可执行映像文件各个段真正执行时在存储器中的位置关系。

扩展： *虽然这部分知识较难理解，但是如果读者想真正把链接、分散加载技术学懂，推荐读者阅读《ARM® Developer Suite——Linker and Utilities Guide》，该书对这部分知识进行了详细的讲解。在此处的讨论的前提是，默认读者对链接方面的知识有个大概的了解。*

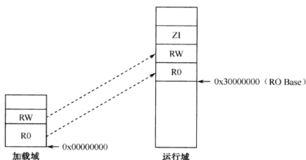


图 7-17　程序加载域和运行域的分布情况

前文讲到，从 NOR FASLH 启动后，当启动代码执行到第 109 行时，会跳转到此处执行。此时，程序加载域和运行域的分布情况如图 7-17 所示。

加载域的起始地址是 0x00000000，为什么呢？因为加载域反映了 ARM 可执行映像文件各个段存放在存储器中时的位置关系，程序下载到 NOR FLASH 中，又知道 NOR FALSH 的起始地址是 0x00000000，因此加载域的起始地址是 0x00000000，在加载域中，首先存放 RO，后面紧跟着存放的是 RW，并没有 ZI。RO 为代码段，属性为只读；RW 为已经初始化的全局变量段，属性为可读、可写；ZI 为未初始化的全局变量段。为什么在加载域中没有 ZI 呢，这主要是为了减小 ARM 可执行映像文件的容量，因为 ZI 中存放的是未初始化的全局变量，那么只需要记录该段的容量即可，在运行域中需要根据这个容量分配内存，然后用 0 填充。

运行域的起始地址是 0x30000000，又是哪里来的呢？

如图 7-21 所示，在“ARM Linker”选项中有一项是“RO Base”，这个地方就表明了运行域中 RO 的起始地址。由图 7-18 可以看到，“RW Base”为空，这表明在运行域中 RW 紧跟着 RO 的后面。

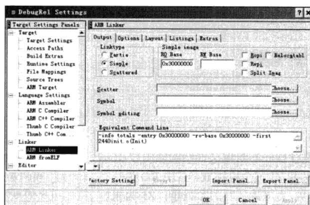


图 7-18　指定 RO Base

启动代码做的工作就如图 7-20 中虚线部分所指示的，将各个段搬移到指定的位置，然后将 ZI 初始化为 0。下面这段程序就是围绕这个工作来展开。

```
116  copy_proc_beg
117        adr        r0,   ResetEntry
118        ldr        r2,   BaseOfROM
119        cmp        r0,   r2
120        ldreq      r0,   TopOfROM
121        beq        InitRam
122        ldr r3,    TopOfROM
123  0
124        ldmia      r0!,    {r4-r7}
125        stmia      r2!,    {r4-r7}
126        cmp        r2,     r3
127        bcc        %B0

128        sub        r2,     r2,   r3
129        sub        r0,     r0,   r2
```

有了上面的讲解，下面在看第 116～129 行程序即可很容易理解。在上述程序中，BaseOfROM 和 TopOfROM 是用 DCD 分配的内存单元，里面存放的是 RO 的起始地址和结束地址（准确点说是结束地址+1）。该起始地址和结束地址是由编译器自动生成的，在程序中使用即可，见第 183～184 行。

第 117 行，使用 adr 指令将 ResetEntry 的相对地址加载到寄存器 r0 中，这个地址就是 0。

第 118 行，使用 ldr 指令将 RO 的起始地址加载到寄存器 r2 中。

第 119 行，比较 r0 和 r2 是否相同，如图 7-19 所示，可以看出 r0 和 r2 并不相等，因此第 119～120 行并不执行。

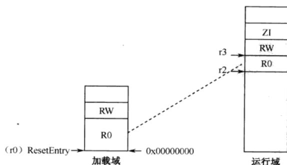


图 7-19　RO 的搬移

第 122 行，使用 ldr 指令将 RO 的结束地址+1 加载到寄存器 r3 中。

执行完上述指令后，图 7-22 详细展示程序执行的最后结果，下面的工作就是将代码从 r0 指示的加载域中的 RO 搬移到由 r2、r3 所限定的运行域中的 RO。

第 123 行，定义了一个局部标号 0。

第 124 行，使用批量加载指令 ldmia，将 r0 地址处的数据加载到寄存器 r4～r7 中，注意后面的感叹号“！”说明取完数据后，r0 的地址自动更新，即指向下一个地址处。此外，可以得出这样的结论，每次搬移 16 个字节（每个寄存器是 4 个字节的长度，共 4 个寄存器）。

第 125 行，将 r4～r7 中的数据存储到 r2 开始的地址处，同时 r2 地址自动更新。

第 126～127 行，比较 r2 和 r3 的值。如果 r2 的值小于 r3 的值，就跳转到第 123 行定义的局部标号 0 处执行。更形象的理解是，如图 7-22 所示，当 r2 小于 r3 时，说明 RO 还没有搬移完，因此就接着搬移，直到 r2 的值大于 r3 的值。

第 128～129 行，这两行的作用就是调整 r0 的值，使其指向加载域中 RW 的起始地址。为了更形象地展示这一调整的过程，请读者参见图 7-20。

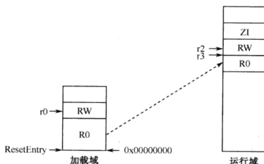


图 7-20　调整 r0 的值，使其指向 RW 的首地址

由第 124 行的讨论可知，每次搬移 16 个字，因此最后可能出现的情况如图 7-20 所示，当 r2 的值大于 r3 的值时，停止搬移，那么现在的情况是已经完成了 RO 从加载域到运行域的搬移，下面需要完成 RW 从加载域到运行域的搬移。面临的首要问题是如何在加载域中找到 RW 的起始地址，现在知道的信息是在加载域中 RW 紧邻着 RO 存放。由图 7-20 可以看到，r0 已经移到了 RW，只要计算出这个偏移，即可计算出 RW 的首地址。这个偏移地址怎么计算呢，其实就是 r2～r3，读者可以慢慢地理解。

第 128 行计算出偏移地址 r2～r3，将其存放在寄存器 r2 中。

第 129 行，将 r0 的值减去这个偏移地址，即得到了 RW 的起始地址。

扩展提示：其实，在这种加载域和运行域布局情况下，即加载域和运行域中 RO 和 RW 紧挨着存放，完全可以将 RO 和 RW 整体搬移，但是该启动代码照顾到更为一般的情况，并没有采取这种搬移方法，而是采取各个段分别搬移的方法。

```
130 InitRam
131         ldr        r2,   BaseOfBSS
132         ldr        r3,   BaseOfZero
133 0
134         cmp        r2,   r3
135         ldrcc      r1,   [r0],   #4
136         strcc      r1,   [r2],   #4
137         bcc        %b0
```

有了前面 RO 搬移的讲解，下面 RW 的搬移工作变得较为简单。在 RO 搬移完以后，r0 已经指向了加载域中 RW 的起始地址处。

第 131 行，将 r2 指向运行域中 RW 的起始地址处，其中 BaseOfBSS 是在 185 行定义的。

第 132 行，将 r3 指向 RW 的结束地址+1 处，又因为 ZI 在 RW 后面紧挨着存放，所以可以得出这样的结论：RW 的结束地址+1 就是 ZI 的起始地址。

RW 的搬移如图 7-21 所示。

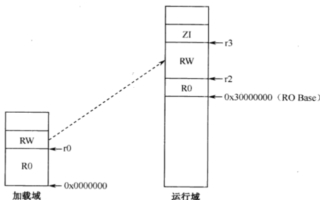


图 7-21　RW 的搬移

第 133 行定义了一个局部标号 0。

第 134 行比较 r2 和 r3 的值。

第 135 行，ldr 指令后面的 cc 表示条件执行，如果 r2 的值小于 r3 的值就从 r0 地址处取出一个字数据，加载到 r1 中，然后 r1 自动加 4，即 r1=r1+4。

第 135 行，将 r1 中的数据存储到 r2 指向的地址处，然后，r2 的值自动加 4，即 r2=r2+4。

第 137 行，如果 r2 的值小于 r3，则跳转到第 133 行定义的局部标号 0 处执行。

更形象地理解这个搬移过程，可以借助下面这个类 C 语言伪码理解：

```
do
{
    r2 指向的存储单元 = r0 指向的存储单元中的值;
    r2 = r2 +4 ;
    r0 = r0 +4 ;
}
while(r2 < r3) ;
```

经过前面的讲解，已经实现了 RO 和 RW 的搬移工作，下面需要做的就是将 ZI 初始化为 0 即可。注意，这里并不是 ZI 的搬移，因为在加载域中没有 ZI，所以不存在搬移这一说法。

```
138         mov         r0,  #0
139         ldr         r3,  EndOfBSS
140  1
141         cmp         r2,  r3
```

```
142        strcc      r0,    [r2],   #4
143        bcc        %b1
```

RW 搬移结束后，r2 指向了 RW 的结束地址+1 处，即 ZI 的起始地址处。

第 138 行，将 0 赋值给寄存器 r0。

第 139 行，将 ZI 的结束地址+1 加载到寄存器 r3，即 r3 指向了 ZI 的结束地址+1 处。程序执行到现在，加载域中的分布情况如图 7-22 所示，可以看出，r2 指向了 ZI 的起始地址，r3 指向了 ZI 的结束地址+1 处。

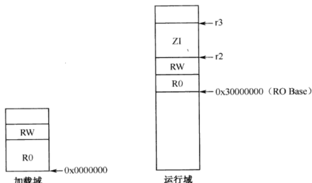


图 7-22　初始化 ZI

第 140～143 行就是将 ZI 用 0 填充，即实现了初始化为 0。

可以说，到此为止，启动代码所做的工作已经接近尾声，下面回顾一下启动代码将程序从加载域到运行域的搬移过程。

（1）找到加载域中各个段的起始地址和结束地址。

方法：用 adr 指令找到了 RO 的起始地址。

（2）找到运行域中各个段的起始地址和结束地址。

方法：使用编译器自动生成的各个段的起始地址和结束地址。

（3）使用循环执行搬移即可。

```
; Setup IRQ handler
144        ldr   r0,     =HandleIRQ   ;This routine is needed
145        ldr   r1,     =IsrIRQ
146        str   r1,     [r0]
```

对于第 144～146 行，读者可以结合第 47 行理解，这里实现的功能是安装中断向量表。

```
147        b      Main     ;Do not use main() because ......
```

第 147 行使用 b 跳转指令跳转到 C 语言函数 Main 处执行，这里主要有如下注释：在自己定义的 C 语言函数中，主函数名不能用 main，为什么呢？原因是当读者使用 ARM C 库实现程序加载域到运行域的初始化时，默认会跳到 main 处。因此，如果不使用 ARM C 库开发时，就不能使用 main 函数。

```
148 InitStacks
149         mrs  r0, cpsr
150         bic  r0, r0, #MODEMASK
151         orr  r1, r0, #UNDEFMODE|NOINT
152         msr  cpsr_cxsf, r1                ;UndefMode
153         ldr  sp, =UndefStack              ;UndefStack=0x33FF5C00

154         orr  r1, r0, #ABORTMODE|NOINT
155         msr  cpsr_cxsf, r1                ;AbortMode
156         ldr  sp, =AbortStack              ;AbortStack=0x33FF6000

157         orr  r1, r0, #IRQMODE|NOINT
158         msr  cpsr_cxsf, r1                ;IRQMode
159         ldr  sp, =IRQStack                ;IRQStack=0x33FF7000

160         orr  r1, r0, #FIQMODE|NOINT
161         msr  cpsr_cxsf, r1                ;FIQMode
162         ldr  sp, =FIQStack                ;FIQStack=0x33FF8000

163         bic  r0, r0, #MODEMASK|NOINT
164         orr  r1, r0, #SVCMODE
165         msr  cpsr_cxsf, r1                ;SVCMode
166         ldr  sp, =SVCStack                ;SVCStack=0x33FF5800
167         mov  pc, lr
168         ltorg
```

上述程序是初始化堆栈，在第 3 章已经讲解过，这段程序是在第 105 行进行的调用，第 167 行实现了函数调用后的返回。

```
169 SMRDATA
170         DCD 0x22011000
171         DCD 0x00000700      ;GCS0
172         DCD 0x00000700      ;GCS1
173         DCD 0x00000700      ;GCS2
174         DCD 0x00000700      ;GCS3
175         DCD 0x00000700      ;GCS4
176         DCD 0x00000700      ;GCS5
177         DCD ((B6_MT<<15)+(B6_Trcd<<2)+(B6_SCAN))      ;GCS6
178         DCD ((B7_MT<<15)+(B7_Trcd<<2)+(B7_SCAN))      ;GCS7
179         DCD ((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Tsrc<<18)+REFCNT)
180         DCD 0xB1
181         DCD 0x30            ;MRSR6 CL=3clk
182         DCD 0x30            ;MRSR7 CL=3clk
```

第 169～182 是初始化存储器控制器时的一些参数，对此读者可以不必细究，讲解到存储器控制器时会详细讲解各个参数的具体含义。

```
183 BaseOfROM     DCD     |Image$$RO$$Base|
184 TopOfROM      DCD     |Image$$RO$$Limit|
185 BaseOfBSS     DCD     |Image$$RW$$Base|
186 BaseOfZero    DCD     |Image$$ZI$$Base|
187 EndOfBSS      DCD     |Image$$ZI$$Limit|
```

第 183～187 行，将编译器自动生成的几个参数进行引用，主要是为了编程方便，读者完全可以直接使用这些参数，在使用之前需要先使用 IMPORT 关键字，将定义的这几个参数引入该源文件中，如第 28～32 行展示了 IMPORT 关键字的使用方法。

```
188         ALIGN
189 AREA RamData,   DATA,   READWRITE

190         ^   _ISR_STARTADDRESS           ; _ISR_STARTADDRESS=0x33FF_FF00
191 HandleReset         #   4
192 HandleUndef         #   4
193 HandleSWI           #   4
194 HandlePabort        #   4
195 HandleDabort        #   4
196 HandleReserved      #   4
197 HandleIRQ           #   4
198 HandleFIQ           #   4

;IntVectorTable   ;@0x33FF_FF20
199 HandleEINT0         #   4
200 HandleEINT1         #   4
201 HandleEINT2         #   4
202 HandleEINT3         #   4
203 HandleEINT4_7       #   4
204 HandleEINT8_23      #   4
205 HandleCAM           #   4
206 HandleBATFLT        #   4
207 HandleTICK          #   4
208 HandleWDT           #   4
209 HandleTIMER0        #   4
210 HandleTIMER1        #   4
211 HandleTIMER2        #   4
212 HandleTIMER3        #   4
213 HandleTIMER4        #   4
214 HandleUART2         #   4
215 HandleLCD           #   4                   ;@0x33FF_FF60
```

```
216  HandleDMA0        #    4
217  HandleDMA1        #    4
218  HandleDMA2        #    4
219  HandleDMA3        #    4
220  HandleMMC         #    4
221  HandleSPI0        #    4
222  HandleUART1       #    4
223  HandleNFCON       #    4
224  HandleUSBD        #    4
225  HandleUSBH        #    4
226  HandleIIC         #    4
227  HandleUART0       #    4
228  HandleSPI1        #    4
229  HandleRTC         #    4
230  HandleADC         #    4
231      END
```

第 189 行定义了一个数据段。

第 190～230 是在内存中定义了一个内存表，用于存放各种外部中断的中断处理程序的入口地址，在第 3 章中进行过讲解，同时读者可以结合图 7-5、图 7-6 和图 7-7 理解。

内存表分布情况如表 7-5 所示。

表 7-5　内存表分布情况

启动代码中的标号	内 存 地 址	数　据
HandleReset	0x33FFFF00	
HandleUndef	0x33FFFF04	
HandleSWI	0x33FFFF08	
HandlePabort	0x33FFFF0C	
HandleDabort	0x33FFFF10	
HandleReserved	0x33FFFF14	
HandleIRQ	0x33FFFF18	
HandleFIQ	0x33FFFF1C	
HandleEINT0	0x33FFFF20	
HandleEINT1	0x33FFFF24	
HandleEINT2	0x33FFFF28	
HandleEINT3	0x33FFFF2C	
HandleEINT4__7	0x33FFFF30	
HandleEINT8__23	0x33FFFF34	
HandleCAM	0x33FFFF38	
HandleBATFLT	0x33FFFF3C	
HandleTICK	0x33FFFF34	
HandleWDT	0x33FFFF38	

（续表）

启动代码中的标号	内存地址	数　据
……	……	
HandleADC	0x33FFFF80	

7.3 启动代码主要功能模块分析

在前面的分析中，主要对一个常见的启动代码进行了详细的分析，并没有对各功能功模块进行探讨。在本节中，将针对启动代码各个功能模块进行宏观的分析与讲解，尽量避开启动代码的限制，力图从宏观的角度讲解启动代码所完成的功能以及各个功能模块的具体实现细节。

在基于 ARM 处理器的嵌入式系统开发中，应用程序大多采用 C 或者 C++等高级语言编写。在运行应用程序之前，需要对系统进行初始化，因此在系统上电后需要有一段引导程序完成对系统资源的初始化，为用户程序建立基本的运行环境。

ARM 公司只开发 ARM 的内核，其他公司在获得 ARM 的内核后自行生产自己的 SoC 芯片，所以不同厂家生产的芯片，启动代码也不尽相同，但是启动代码大都实现以下功能。

- 建立异常中断向量表。
- 初始化各模式的堆栈。
- 初始化硬件。
- 初始化应用程序执行环境。
- 最后跳转到主应用程序。

7.3.1 建立中断向量表

1．硬件固有的异常中断向量表

系统上电后，程序将从 0 地址处开始执行，因此在系统初始化阶段必须保证在 0 地址处存在正确的代码，即要求在 0 地址处的存储器是非易失性的 ROM 或 Flash。中断向量表就存储在这个非易失性的 ROM 或 Flash 里面。一旦异常中断发生，ARM 处理器便强制把 PC 指针指向异常中断向量表中对应的中断类型的具体地址，然后从向量表跳转到存储器里存放异常中断服务程序的地址，执行具体的异常中断服务程序。

地址	指令
0x0000001c	b　HandlerFIQ
0x00000018	b　HandlerIRQ
0x00000014	b
0x00000010	b　HandlerDabort
0x0000000C	b　HandlerPabort
0x00000008	b　HandlerSWI
0x00000004	b　HandlerUndef
0x00000000	b　ResetHandler

图 7-23　异常中断向量表

ARM 要求异常中断向量表必须存放在从 0 地址开始的连续 8×4 字节的空间内，如图 7-23 所示，因为每种中断只占据向量表中 1 个字的存储空间。因此，一般情况下，异常中断向量表中 0x00～0x1c 这段程序只是简单地包含跳转指令，程序从此处跳转到具体的中断处理函数去执行。

具体实现程序如下：

```
AREA Init,CODE,READONLY
    ENTRY
ResetEntry
    b    ResetHandler
    b    HandlerUndef

    b    HandlerSWI

    b    HandlerPabort

    b    HandlerDabort

    b    .

    b    HandlerIRQ

    b    HandlerFIQ
```

2．软件设定的异常中断向量表

S3C2440 处理器还规定了 EINT0 到 INT_ADC 32 个中断源，用户可以通过控制中断模式寄存器 INTMODE 来定义某个中断是属于 IRQ 还是 FIQ。当这 32 个中断源中某一个发出中断请求时，CPU 都跳转到 b HandlerIRQ（假设此时所有中断均设为 IRQ 模式），为了能确定到底是哪一个中断发生，需要在 RAM 区开辟一段内存空间，建立一张软件设定的中断向量表（如表 7-6 所示），用来存放各个中断服务程序的入口地址，每个中断服务程序的入口地址都占用一个表项，1 个字的空间。在应用程序中，将中断服务程序入口地址写入相应的表项空间内，完成中断向量的设置。通常，用_ISR _STARTADDRESS 表示该中断向量表的起始地址，一般将其放在 RAM 的最后一段地址空间。

表 7-6　软件设定的中断向量表

内 存 地 址	中断服务程序入口
_ISR _STARTADDRESS	HandleReset
_ISR _STARTADDRESS+4	HandleUndef
_ISR _STARTADDRESS+8	HandleSWI
⋮	⋮
_ISR _STARTADDRESS+31*4	HandleADC

以响应外部中断 0 为例，其具体中断服务程序入口地址为 Eint0Isr 的示意程序如下：

```
ldr pc, = HandlerEINT0
…
HandlerEINT0  HANDLER  HandleEINT0
```

其中，HandleEINT0 为 EINT0 中断源在软件设定的中断向量表中的相应表项的地址。在该表项中，事先存有 EINT0 具体中断服务程序入口地址（Eint0Isr）。HANDLER 是一个

宏，它将 HandleEINT0 指向的内容值赋给 PC，即将 EINT0 具体中断服务程序入口地址送给 PC，这样就完成了向特定中断服务程序的转移，其宏定义程序如下：

```
    MACRO
$HandlerLabel HANDLER $HandleAddr
$HandlerLabel
    sub     sp,sp,#4
    stmfd   sp!,{r0}
    ldr     r0,=$HandleAddr
    ldr     r0,[r0]
    str     r0,[sp,#4]
    ldmfd   sp!,{r0,pc}
    MEND
```

其中，HANDLER 为宏名，$HandleAddr 为宏形式参数。在上面的宏调用中，用宏实际参数 HandleEINT0 替换宏形式参数$HandleAddr。

7.3.2 初始化各个模式的堆栈

ARM 处理器有 7 种工作模式，如表 7-7 所示。

表 7-7 ARM 处理器的 7 种工作模式

处理器模式	说　明
用户模式（User）	正常程序执行的模式
快速中断模式（FIQ）	用于高速数据传输和通道处理
外部中断模式（IRQ）	用于通常的中断处理
特权模式（Supervisor）	供操作系统使用的一种保护模式
数据访问终止模式（Abort）	用于虚拟存储及存储保护
未定义指令终止（Undefined）	用于支持通过软件仿真硬件的协处理器
系统模式（System）	用于运行特权级的操作系统任务

ARM 处理器系统模式和用户模式公用同一块堆栈空间，其他 5 种模式都有自己的堆栈寄存器 SP，保存其堆栈指针，因此其堆栈需要分别初始化。初始化方法是：改变状态寄存器 CPSR 内的状态位，使处理器切换到不同的状态，然后给 SP 赋值，对 CPSR 寄存器的操作采取“读取—修改—写入”的方式。需要注意的是，不要切换到用户模式进行用户模式堆栈的初始化，因为在用户模式下是不能对 CPSR 寄存器相应的模式控制位进行写操作的。初始化堆栈的程序如下：

```
InitStacks
    mrs  r0,cpsr
    bic  r0,r0,#MODEMASK
    ;初始化未定义指令终止模式堆栈
    orr  r1,r0,#UNDEFMODE|NOINT
    msr  cpsr_cxsf,r1
```

```
ldr   sp,=UndefStack
;初始化数据访问终止模式堆栈
orr   r1,r0,#ABORTMODE|NOINT
msr   cpsr_cxsf,r1
ldr   sp,=AbortStack
;初始化外部中断模式堆栈
orr   r1,r0,#IRQMODE|NOINT
msr   cpsr_cxsf,r1
ldr   sp,=IRQStack
;初始化快速中断模式堆栈
orr   r1,r0,#FIQMODE|NOINT
msr   cpsr_cxsf,r1
ldr   sp,=FIQStack
;初始化系统模式堆栈
orr   r1,r0,#SYSMODE|NOINT
msr   cpsr_cxsf,r1
ldr   sp,=UserStack
;初始化特权模式堆栈
bic   r0,r0,#MODEMASK|NOINT
orr   r1,r0,#SVCMODE
msr   cpsr_cxsf,r1
mov   pc,lr
```

7.3.3 初始化系统硬件

在系统启动阶段对硬件初始化主要涉及以下几个方面。

- 关闭看门狗。
- 屏蔽所有中断。
- 初始化 PLL 和系统时钟。PLL 的输出频率就是处理器内核的工作频率。
- 初始化存储系统。S3C2440 处理器使用 13 个专用的特殊功能寄存器来控制外部存储器的读/写操作，通过对这 13 个特殊功能寄存器编程，可以设定外部数据总线宽度、访问周期、定时的控制信号等参数。此外要特别注意，当系统外接内存芯片时，要根据具体芯片的时序参数配置相关寄存器，以确保内存芯片的性能发挥到最佳，尽量提高程序的执行速度。

7.3.4 初始化应用程序的执行环境并跳转到主程序执行

一个简单的可执行程序的映像文件结构如图 7-24 所示，它由 RO、RW 和 ZI 三个段组成。其中，RO 为代码和只读数据段；RW 为可读/写的数据段；ZI 为未初始化的数据段。当从 NAND FLASH 启动时，需将 RW 复制到 SDRAM 中，并将 ZI 清零。

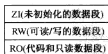


图 7-24 ARM 映像文件结构（加载域）

S3C2440 处理器支持两种启动方式：从 NOR

FLASH 启动和从 NAND FLASH 启动。

NOR FLASH 可以像 SDRAM 一样随机访问，并且读取速度快，既可以存储程序又可以运行程序，但是 NOR FLASH 的价格比较贵。

NAND FLASH 容量大，价格低廉，广泛用于嵌入式系统中。但是，NAND FLASH 的随机读取速度慢，只能用来存储程序，无法运行程序。

S3C2440 处理器内部集成了 NAND FLASH 控制器，上电复位后，处理器通过内部硬件电路实现将 NAND FLASH 前 4KB 的内容复制到片内 RAM，这样就可以实现从 NAND FLASH 启动，但是当代码容量大于 4 KB 时，需要将代码复制到 SDRAM 中执行，此时需要调用对 NAND FLASH 进行读/写的函数来实现。如果在链接的时候使用 ro-base 和 rw-base 参数，那么编译器会自动生成特殊的符号标记各段的起始地址和结束地址，如表 7-8 所示。

表 7-8　编译器生成的各段标号

编译器生成的符号	含　义
|Image$$RO$$Base|	RO 起始地址
|Image$$RO$$Limit|	RO 结束地址+1
|Image$$RW$$Base|	RW 起始地址
|Image$$RW$$Limit|	RW 结束地址+1
|Image$$ZI$$Base|	ZI 起始地址
|Image$$ZI$$Limit|	ZI 结束地址+1

因此，在启动代码中可以使用这些标号来实现代码的搬移工作。以下代码主要完成将代码从加载域搬移到运行域，并将 ZI 清零。

```
BaseOfROM       DCD     |Image$$RO$$Base|
TopOfROM        DCD     |Image$$RO$$Limit|
BaseOfBSS       DCD     |Image$$RW$$Base|
BaseOfZero      DCD     |Image$$ZI$$Base|
EndOfBSS        DCD     |Image$$ZI$$Limit|
…
copy_proc_beg
    adr         r0, ResetEntry
    ldr         r2, BaseOfROM
    cmp         r0, r2                  ;查看运行域和加载域是否相同
    ldreq       r0, TopOfROM
    beq         InitRam
    ldr         r3, TopOfROM
0
    ldmiar      0!, {r4-r7}
    stmia       r2!, {r4-r7}
    cmp         r2, r3
```

```
        bcc       %B0

        sub       r2, r2, r3
        sub       r0, r0, r2

InitRam
        ldr       r2, BaseOfBSS
        ldr       r3, BaseOfZero
0
        Cmp       r2, r3                    ;将 ZI 清零
        ldrcc     r1, [r0], #4
        strcc     r1, [r2], #4
        bcc       %B0

        mov       r0,   #0
        ldr       r3,   EndOfBSS
1
        cmp       r2,   r3
        strcc     r0, [r2], #4
        bcc       %B1
```

7.3.5 跳转到 C 语言主程序执行

当系统初始化工作完成后，就需要跳转到主应用程序，有两种方法可以实现此功能。

第 1 种是使用 b 跳转指令跳转到用户自定义的主函数。这时，函数名可由用户自己定义。例如，Main 就可以使用 b Main 跳转到主应用程序。

第 2 种是调用标准 C 库函数__main()。调用__main()将大大简化汇编启动代码的编写，将代码从加载域复制到运行域，以及 ZI 的清零工作都由 C 库来完成。尤其是使用分散加载机制来实现复杂的存储器映射时，__mian()可以直接调用_rt_entry()，用于建立 C 库运行所必需的环境。

注意： 在第 2 种情况下，用户主应用程序的函数名必须是 mian()。此外，__rt_enty()并不是 C 函数，而是使用 ARM C 库编程的起始点，__rt_entry()不能用 C 语言实现，因为这时堆栈还没有建立，堆栈由__rt_inital_stackheap()来建立，因此还需要重写__rt_inital_stackheap()函数，可用如下程序实现：

```
EXPORT __rt_ inital _ stackheap
__rt_ inital _ stackheap
LDR R0, =|Image$$ZI$$Limit|
LDR R1, =|Image$$ZI$$Limit|+0x4000
MOV PC,LR
```

本文中使用的是第 1 种方法，有兴趣的读者可以尝试使用第 2 种方法。

7.4 本章小结

在本章中，重点对启动代码进行了讲解，主要是向读者展示了在执行用户的C语言应用程序之前都做了哪些工作以及是如何使用汇编语言来实现这些工作的。此外，对于ARM可执行映像文件的结构并没有进行具体的讲解，读者可以参阅相关书籍，推荐读者阅读《ARM® Developer Suite——Linker and Utilities Guide》。

此外，代码从加载域到运行域的搬移是个重点，学习ARM的重点就在于此，但是目前这方面的书籍较少，推荐有能力的读者阅读《程序员的自我修养——链接、装载与库》（编者：俞甲子、石凡、潘爱民）。

7.5 本章附录——完整版启动代码

为了读者查阅方便，将完整的启动代码展示在这里。

```
1          GET option.inc
2          GET memcfg.inc
3          GET 2440addr.inc
; constants defination
4    USERMODE          EQU        0x10
5    FIQMODE           EQU        0x11
6    IRQMODE           EQU        0x12
7    SVCMODE           EQU        0x13
8    ABORTMODE         EQU        0x17
9    UNDEFMODE         EQU        0x1b
10   MODEMASK          EQU        0x1f
11   NOINT             EQU        0xc0

     ;The location of stacks
12   UserStack     EQU    (_STACK_BASEADDRESS-0x3800)      ;0x33ff4800 ~
13   SVCStack      EQU    (_STACK_BASEADDRESS-0x2800)      ;0x33ff5800 ~
14   UndefStack    EQU    (_STACK_BASEADDRESS-0x2400)      ;0x33ff5c00 ~
15   AbortStack    EQU    (_STACK_BASEADDRESS-0x2000)      ;0x33ff6000 ~
16   IRQStack      EQU    (_STACK_BASEADDRESS-0x1000)      ;0x33ff7000 ~
17   FIQStack      EQU    (_STACK_BASEADDRESS-0x0)         ;0x33ff8000 ~

18         MACRO
19   $HandlerLabel       HANDLER $HandleAddr
20   $HandlerLabel
21         sub       sp,   sp,   #4
22         stmfd     sp!,  {r0}
```

```
23        ldr       r0,  =$HandleAddr
24        ldr       r0,  [r0]
25        str       r0,  [sp, #4]
26        ldmfd     sp!, {r0, pc}
27        MEND

28        IMPORT    |Image$$RO$$Base|       ; Base of ROM code
29        IMPORT    |Image$$RO$$Limit|      ; End of ROM code (=start of ROM data)
30        IMPORT    |Image$$RW$$Base|       ; Base of RAM to initialise
31        IMPORT    |Image$$ZI$$Base|       ; Base and limit of area
32        IMPORT    |Image$$ZI$$Limit|      ; to zero initialise
33        IMPORT    Main                    ; The main entry of mon program
34        IMPORT    RdNF2SDRAM              ; Copy Image from Nand Flash to SDRAM


35        AREA      Init, CODE, READONLY
36        ENTRY
37    ResetEntry
38        b     ResetHandler
39        b     HandlerUndef        ;handler for Undefined mode
40        b     HandlerSWI          ;handler for SWI interrupt
41        b     HandlerPabort       ;handler for PAbort
42        b     HandlerDabort       ;handler for DAbort
43        b     .                   ;reserved
44        b     HandlerIRQ          ;handler for IRQ interrupt
45        b     HandlerFIQ          ;handler for FIQ interrupt

46    HandlerFIQ        HANDLER HandleFIQ
47    HandlerIRQ        HANDLER HandleIRQ
48    HandlerUndef      HANDLER HandleUndef
49    HandlerSWI        HANDLER HandleSWI
50    HandlerDabort     HANDLER HandleDabort
51    HandlerPabort     HANDLER HandlePabort
52    IsrIRQ
53        sub       sp, sp, #4          ;reserved for PC
54        stmfd     sp!, {r8-r9}
55        ldr       r9, =INTOFFSET
56        ldr       r9, [r9]
57        ldr       r8, =HandleEINT0
58        add       r8, r8, r9, lsl #2
59        ldr       r8, [r8]
60        str       r8, [sp, #8]
61        ldmfd     sp!, {r8-r9, pc}
```

```
62          LTORG

63    ResetHandler
64          ldr   r0, =WTCON                  ;watch dog disable
65          ldr   r1, =0x0
66          str   r1, [r0]

67          ldr   r0, =INTMSK
68          ldr   r1, =0xffffffff             ;all interrupt disable
69          str   r1, [r0]
70          ldr   r0, =INTSUBMSK
71          ldr   r1, =0x7fff                 ;all sub interrupt disable
72          str   r1, [r0]

73          ldr   r0, =LOCKTIME
74          ldr   r1, =0xffffff
75          str   r1, [r0]
76          ldr   r0, =CLKDIVN
77          ldr   r1, =CLKDIV_VAL
78          str   r1, [r0]

79          [ CLKDIV_VAL>1                    ; means Fclk:Hclk is not 1:1.
80          mrc  p15, 0, r0, c1, c0, 0
81          orr   r0, r0, #0xc0000000
82          mcr  p15, 0, r0, c1, c0, 0
83          |
84          mrc p15, 0, r0, c1, c0, 0
85          bic r0, r0, #0xc0000000
86          mcr p15, 0, r0, c1, c0, 0
87          ]
     ;Configure UPLL
88          ldr   r0, =UPLLCON
     ;Fin = 12.0MHz,   UCLK = 48MHz
89          ldr   r1, =((U_MDIV<<12)+(U_PDIV<<4)+U_SDIV)
90          str   r1, [r0]
91          nop        ; at least 7-clocks delay must be inserted for setting hardware be completed.
92          nop
93          nop
94          nop
95          nop
96          nop
97          nop
```

```
        ;Configure MPLL
    98      ldr   r0, =MPLLCON
        ;Fin = 12.0MHz,   FCLK = 200MHz
    99      ldr   r1, =((M_MDIV<<12)+(M_PDIV<<4)+M_SDIV)
    100     str   r1, [r0]
        ;Set memory control registers
    101     adrl        r0, SMRDATA         ;please caution!!
    102     ldmia       r0, {r1-r13}
    103     ldr         r0, =BWSCON
    104     stmia       r0, {r1-r13}
    105     bl    InitStacks
;**********************************************************************
    106     ldr         r0,  =BWSCON
    107     ldr         r0,  [r0]
    108     ands        r0,  r0,  #6            ;OM[1:0] != 0,   NOR FLash boot
    109     bne         copy_proc_beg           ;do not read nand flash
    110     adr         r0,  ResetEntry         ;OM[1:0] == 0,   NAND FLash boot
    111     cmp         r0,  #0                 ;if use Multi-ice,
    112     bne         copy_proc_beg           ;do not read nand flash for boot

;**********************************************************************
    113 nand_boot_beg
    114     bl    RdNF2SDRAM
    115     ldr   pc,   =copy_proc_beg
;**********************************************************************
    116 copy_proc_beg
    117     adr         r0,  ResetEntry
    118     ldr         r2,  BaseOfROM
    119     cmp         r0,  r2
    120     ldreq       r0,  TopOfROM
    121     beq         InitRam
    122     ldr r3,     TopOfROM
    123 0
    124     ldmia       r0!,   {r4-r7}
    125     stmia       r2!,   {r4-r7}
    126     cmp         r2,    r3
    127     bcc         %B0

    128     sub         r2,  r2,  r3
    129     sub         r0,  r0,  r2

    130 InitRam
    131     ldr         r2,  BaseOfBSS
```

```
132        ldr        r3，  BaseOfZero
133  0
134        cmp        r2，  r3
135        ldrcc      r1，  [r0]，  #4
136        strcc      r1，  [r2]，  #4
137        bcc        %b0
138        mov        r0，  #0
139        ldr        r3，  EndOfBSS
140  1
141        cmp        r2，  r3
142        strcc      r0，  [r2]，  #4
143        bcc        %b1

     ; Setup IRQ handler
144        ldr   r0，=HandleIRQ        ;This routine is needed
145        ldr   r1，=IsrIRQ           ;if there is not 'subs pc，lr，#4' at 0x18，  0x1c
146        str   r1，[r0]

147        b     Main;Do not use main() because ......

148  InitStacks
149        mrs   r0，cpsr
150        bic   r0，r0，#MODEMASK
151        orr   r1，r0，#UNDEFMODE|NOINT
152        msr   cpsr_cxsf，r1          ;UndefMode
153        ldr   sp，=UndefStack        ;UndefStack=0x33FF_5C00

154        orr   r1，r0，#ABORTMODE|NOINT
155        msr   cpsr_cxsf，r1          ;AbortMode
156        ldr   sp，=AbortStack        ;AbortStack=0x33FF_6000

157        orr   r1，r0，#IRQMODE|NOINT
158        msr   cpsr_cxsf，r1          ;IRQMode
159        ldr   sp，=IRQStack          ;IRQStack=0x33FF_7000

160        orr   r1，r0，#FIQMODE|NOINT
161        msr   cpsr_cxsf，r1          ;FIQMode
162        ldr   sp，=FIQStack          ;FIQStack=0x33FF_8000
163        bic   r0，r0，#MODEMASK|NOINT
164        orr   r1，r0，#SVCMODE
165        msr   cpsr_cxsf，r1          ;SVCMode
166        ldr   sp，=SVCStack          ; SVCStack=0x33FF_5800
167        mov pc，lr
```

```
168         ltorg

169 SMRDATA
170         DCD 0x22011000
171         DCD 0x00000700     ;GCS0
172         DCD 0x00000700     ;GCS1
173         DCD 0x00000700     ;GCS2
174         DCD 0x00000700     ;GCS3
175         DCD 0x00000700     ;GCS4
176         DCD 0x00000700     ;GCS5
177         DCD ((B6_MT<<15)+(B6_Trcd<<2)+(B6_SCAN))      ;GCS6
178         DCD ((B7_MT<<15)+(B7_Trcd<<2)+(B7_SCAN))      ;GCS7
179         DCD ((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Tsrc<<18)+REFCNT)
180         DCD 0xB1
181         DCD 0x30          ;MRSR6 CL=3clk
182         DCD 0x30          ;MRSR7 CL=3clk

183 BaseOfROM     DCD    |Image$$RO$$Base|
184 TopOfROM      DCD    |Image$$RO$$Limit|
185 BaseOfBSS     DCD    |Image$$RW$$Base|
186 BaseOfZero    DCD    |Image$$ZI$$Base|
187 EndOfBSS      DCD    |Image$$ZI$$Limit|

188         ALIGN

189 AREA RamData， DATA， READWRITE

190         ^    _ISR_STARTADDRESS             ; _ISR_STARTADDRESS=0x33FF_FF00
191 HandleReset          #     4
192 HandleUndef          #     4
193 HandleSWI            #     4
194 HandlePabort         #     4
195 HandleDabort         #     4
196 HandleReserved       #     4
197 HandleIRQ            #     4
198 HandleFIQ            #     4


    ;IntVectorTable    ;@0x33FF_FF20
199 HandleEINT0          #     4
200 HandleEINT1          #     4
201 HandleEINT2          #     4
202 HandleEINT3          #     4
```

```
203  HandleEINT4_7      #    4
204  HandleEINT8_23     #    4
205  HandleCAM          #    4
206  HandleBATFLT       #    4
207  HandleTICK         #    4
208  HandleWDT          #    4
209  HandleTIMER0       #    4
210  HandleTIMER1       #    4
211  HandleTIMER2       #    4
212  HandleTIMER3       #    4
213  HandleTIMER4       #    4
214  HandleUART2        #    4
215  HandleLCD          #    4;@0x33FF_FF60
216  HandleDMA0         #    4
217  HandleDMA1         #    4
218  HandleDMA2         #    4
219  HandleDMA3         #    4
220  HandleMMC          #    4
221  HandleSPI0         #    4
222  HandleUART1        #    4
223  HandleNFCON        #    4
224  HandleUSBD         #    4
225  HandleUSBH         #    4
226  HandleIIC          #    4
227  HandleUART0        #    4
228  HandleSPI1         #    4
229  HandleRTC          #    4
230  HandleADC          #    4
231      END
```

第8章 系统时钟和定时器

系统时钟是整个电路的心脏。了解系统时钟电路的结构对于后面学习定时器、UART等的使用具有重要的作用。总体上来说，与S3C2440处理器有关的时钟主要有4种：Fin、FCLK、HCLK和PCLK。

- Fin即外部输入的晶振频率。
- FCLK主要用于CPU核。
- HCLK主要用在与AHB总线互连的设备（如存储器控制器、LCD控制器、中断控制器以及DMA等）上。
- PCLK主要用在与APB总线互连的低速设备（如定时器、UART、ADC等）上。

8.1 S3C2440时钟系统概述

S3C2440处理器系统时钟分为两部分，外部有时钟输入引脚，内部用2个锁相环将外部输入时钟倍频到处理器工作所需要的时钟。外部时钟频率太高容易受到外部的干扰，因此一般外部时钟频率较低。但是，S3C2440处理器内部工作频率较高，这就需要用锁相环来实现倍频功能，如图8-1所示。虽然锁相环有很多指标，但是读者完全可以将其理解为一个时钟变换电路，低频晶振输入即可得到处理器所使用的较高频率的时钟。

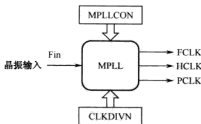


图8-1 锁相环

对于TQ2440开发板，外部晶振的振荡频率是12MHz，经过锁相环MPLL后会输出3种类型的频率：FCLK、HCLK、PCLK。CPU核工作需要FCLK，AHB总线上的外设（如存储器控制器、LCD控制器、中断控制器等）需要使用HCLK，其他低速外设（如ADC、UART、GPIO、RTC、IIC和SPI等）需要PCLK，各个时钟都对应着具体的应用设备，如图8-2所示。

在图8-1中还有两个控制寄存器（MPLLCON和CLKDIVN），分别用于控制分频比。

- MPLLCON控制FCLK和Fin的比例关系。

- CLKDIVN 用于控制 FCLK、HCLK 和 PCLK 之间的比例关系。

此外，S3C2440 内部还有一个锁相环 UPLL，用于将外部时钟倍频到 USB 设备正常工作所需要的时钟频率，工作原理与上面的 MPLL 类似。

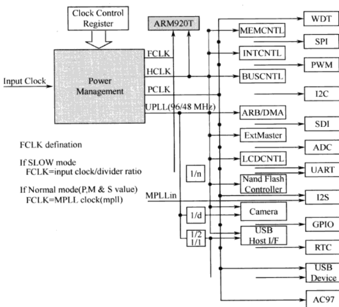


图 8-2　系统时钟分布图

8.1.1　系统时钟初始化

系统时钟初始化流程如图 8-3 所示。

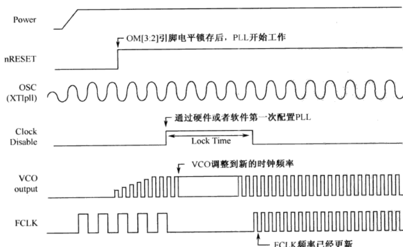


图 8-3　系统时钟初始化流程

系统上电后，S3C2440 处理器会自动锁存 OM3 和 OM2 引脚的电平值，这两个引脚用于选择外部时钟输入方式，如图 8-4 所示。TQ2440 开发板上的 OM3 和 OM2 均接地，即 OM[3:2]=00。所以，时钟源为外部晶振。

Mode OM[3:2]	MPLL State	UPLL State	Main Clock source	USB Clock Source
00	On	On	Crystal	Crystal
01	On	On	Crystal	EXTCLK
10	On	On	EXTCLK	Crystal
11	On	On	EXTCLK	EXTCLK

图 8-4　外部时钟输入方式选择

如图 8-3 所示，系统时钟初始化流程如下。

（1）系统刚上电几毫秒后，FCLK 等于外部晶振的振荡频率，即 FCLK=Fin。

（2）当复位信号 nRESET 恢复高电平后，锁相环按照寄存器 MPLLCON 和 CLKDIVN 设定的倍频比例开始生成所需要的时钟频率。从图 8-3 中可以看到，从锁相环开始工作到输出新的稳定的频率值需要一定的时间（Lock Time，也叫锁相环的捕获时间），经过这段时间后，锁相环输出新的频率值，这是 FCLK 等于锁相环的输出。寄存器 LOCKTIME 中的值对应着图 8-2 中的 Lock Time，初始化时一般将其设为 0xffffff，这是 S3C2440 数据手册上给出的默认值，一般按照这个值初始化 LOCKTIME 寄存器即可满足要求。

（3）经过一段时间后，锁相环 PLL 输出新的时钟频率。

8.1.2　FCLK、HCLK 和 PCLK 与 Fin 的关系

如何控制锁相环 PLL 的输出频率呢？S3C2440 处理器内部有两个寄存器：MPLLCON 寄存器用于控制 FCLK 和 Fin 的比例关系，CLKDIVN 寄存器用于控制 FCLK、HCLK 和 PCLK 之间的比例关系。

FCLK =（2*m*Fin）/（p*2^s）

在上式中，m=MDIV+8，p=PDIV+2，s=SDIV。其中，MDIV、PDIV 和 SDIV 是 MPLLCON 寄存器中的数据，如图 8-5 所示。

MPLLCON	Bit	Description	Initial State
MDIV	[19:12]	Main divider control	0x96/0x4d
PDIV	[9:4]	Pre-divider control	0x03/0x03
SDIV	[1:0]	Post divider control	0x0/0x0

NOTE: When you set MPLL & UPLL values,you have to set the UPLL value first and then the MPLL valuel(Needs intervals approximately 7 NOP).（当设置MPLL & UPLL的值时，需要首先设置UPLL的值，然后在设置MPLL的值，间隔时间大约为7条NOP指令的执行时间。）

图 8-5　MPLLCON 寄存器

注意：在图 8-5 中，可以看到最下面的 NOTE 用来提醒读者注意，在系统初始化阶段应该先初始化 UPLL（USB 时钟），然后等待大约 7 个 nop 指令（空指令）后，再初始化 MPLL。这就是第 7 章讲解启动代码时，第 91～97 行有 7 个 nop 指令的原因。

此外，虽然 PLL 给用户提供了灵活变换系统时钟的功能，但是，并不是在任意的时钟下处理器都能正常工作，基于此种原因，官方给出了系统时钟配置参考，如图 8-6 所示。

Input Frequency	Output Frequency	MDIV	PDIV	SDIV
12.0000 MHz	48.00 MHz(NOTE)	**56(0x38)**	2	2
12.0000 MHz	96.00 MHz(NOTE)	56(0x38)	2	1
12.0000 MHz	271.50 MHz	173(0xad)	2	2
12.0000 MHz	304.00 MHz	68(0x44)	1	1
12.0000 MHz	405.00 MHz	127(0x7f)	2	1
12.0000 MHz	532.00 MHz	125(0x7d)	1	1
16.9344 MHz	47.98 MHz(NOTE)	60(0x3c)	4	2
16.9344 MHz	95.96 MHz(NOTE)	60(0x3c)	4	1
16.9344 MHz	**266.72 MHz**	118(0x76)	2	2
16.9344 MHz	**296.35 MHz**	97(0x61)	1	2
16.9344 MHz	399.65 MHz	110(0x6e)	3	1
16.9344 MHz	530.61 MHz	86(0x56)	1	1
16.9344 MHz	533.43 MHz	118(0x76)	1	1

NOTE: The 48.00 MHz and 96 MHz output is used for UPLLCON register. (48.00 MHz和96.00 MHz的输出频率是UPLLCON寄存器使用的。)

图 8-6　系统时钟配置参考

其中，47.98MHz 和 95.96MHz 主要用于配置 USB 时钟，可以暂时忽略。

下面以第 5 行为例讲解，假设外部晶振输入为 12MHz，MDIV=127，PDIV=1，SDIV=1，则 m=127+8=135，p=2+2=4，s=1。

FCLK =(2*12*135)/(4*2) = 405MHz。

小技巧：由上面的分析可以看到，MDIV、PDIV 和 SDIV 都是占用了 MPLLCON 的某几位，如 PDIV 是 MPLLCON 寄存器的第 4～9 位。因此，初始化时可以使用移位指令来实现对 MPLLCON 的初始化。

例如，已知系统外部晶振输入为 12 MHz，要求 FCLK 输出为 200 MHz，经过计算可以得到 MDIV=92，PDIV=4，SDIV=1。

可以用下面的指令进行初始化（这也是本书中使用的初始化方法，见第 7 章的启动代码第 98～100 行）。

```
M_MDIV      EQU     92      ;Fin=12.0MHz   Fout=200MHz
M_PDIV      EQU     4
M_SDIV      EQU     1       ; 2440A
ldr   r0, =MPLLCON
ldr   r1, =((M_MDIV<<12)+(M_PDIV<<4)+M_SDIV)
str   r1, [r0]
```

下面需要配置 CLKDIVN 寄存器，实现 FCLK、HCLK 和 PCLK 之间的分频比。

在 CLKDIVN 寄存器中，HDIVN 用于控制 FCLK 和 PCLK 的比例关系，如图 8-7 所示，PDIVN 主要用于控制 HCLK 和 PCLK 的比例关系。该分频比的关系可以总结为：

- 当 CLKDIV_VAL=0 时，FCLK:HCLK:PCLK = 1:1:1。
- 当 CLKDIV_VAL=1 时，FCLK:HCLK:PCLK = 1:1:2。
- 当 CLKDIV_VAL=2 时，FCLK:HCLK:PCLK = 1:2:2。
- 当 CLKDIV_VAL=3 时，FCLK:HCLK:PCLK = 1:2:4。
- 当 CLKDIV_VAL=4 时，FCLK:HCLK:PCLK = 1:4:4。
- 当 CLKDIV_VAL=5 时，FCLK:HCLK:PCLK = 1:4:8。

- 当 CLKDIV_VAL=6 时，FCLK:HCLK:PCLK = 1:3:3。
- 当 CLKDIV_VAL=7 时，FCLK:HCLK:PCLK = 1:3:6。

CLKDIVN	Bit	Description	Initial State
DIVN_UPLL	[3]	UCLK Select register(UCLK must be 48MHz for USB) 0:UCLK=UPLL clock 1:UCLK=UPLL clock/2 Set to 0.when UPLL clock is set as 48MHz Set to 1.when UPLL clock is set as 96MHz.	0
HDIVN	[2:1]	00:HCLK=FCLK/1. 01:HCLK=FCLK/2. 10:HCLK=FCLK/4 when CAMDIVN[9]=0. 11:HCLK=FCLK/3 when CAMDIVN[8]=0. HCLK=FCLK/6 when CAMDIVN[8]=1.	00
PDIVN	[0]	0:PCLK has the clock same as the HCLK/1. 1:PCLK has the clock same as the HCLK/2.	0

图 8-7　CLKDIVN 寄存器

在第 7 章的启动代码中选择的是 CLKDIV_VAL=3，即 FCLK:HCLK:PCLK = 1:2:4，结合前面的讲解，可以知道 Fin=12 MHz，FCLK=200 MHz，HCLK=100 MHz，PCLK=50 MHz，最终效果如图 8-8 所示。

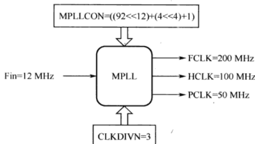


图 8-8　系统时钟初始化最终效果

8.2　定时器原理与应用

下面将具体讲解定时器的原理以及在初始化阶段需要访问的寄存器，最后给出一个具体的定时器实验，复习巩固本节内容。在本节的最后，还对 PWM 功能进行了具体的实验，并且根据示波器实际捕捉到的波形进行了理论和实验的验证。

8.2.1　定时器原理

S3C2440 有 5 个 16 位定时器，定时器 0、1、2 和 3 有脉冲宽度调制（PWM）功能，因此这 4 个定时器也被称为 PWM 定时器。定时器 4 是一个内部定时器，无外部输出引脚。

定时器的时钟源是 PCLK，然后通过内部的分频器分频得到定时器工作所需要的频率。其中，定时器 1、2 公用一个分频器，其他 3 个定时器公用另一个分频器。分频器输入信号经过第 2 级分频器进一步降低时钟频率，然后输出作为定时器工作的时钟。

定时器的内部结构如图 8-9 所示。

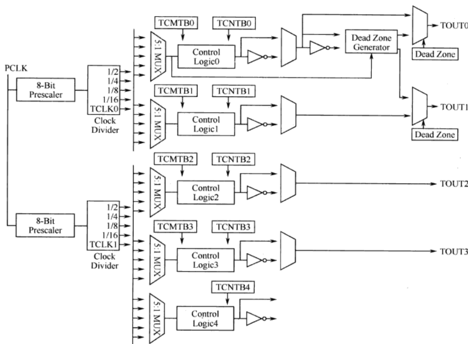


图 8-9　定时器的内部结构

虽然定时器较多，但是工作原理都是相同的，只需要理解一个定时器的工作原理即可。对于某一个定时器，其内部结构原理图如图 8-10 所示。寄存器 TCMPBn 和 TCNTBn 用于缓存定时器 n 的比较值和初始值；TCON 用于控制定时器的开启与关闭；可以通过读取寄存器 TCNTOn 得到定时器的当前计数值。

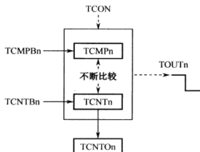


图 8-10　PWM 定时器内部结构原理图

定时器工作原理概述：

- 首先，将定时器的比较值和初始值装入寄存器 TCMPBn 和 TCNTBn 中。
- 然后，设置定时器控制寄存器 TCON，启动定时器。此时，TCMPBn 和 TCNTBn 中的值会加载到寄存器 TCMPn 和 TCNTn 中。
- 此时，定时器会减 1 计数，即 TCNTn 进行减 1 计数，当 TCMPn=TCNTn 时，TOUTn 引脚输出取反。

8.2.2 定时器相关的寄存器

操作定时器的关键是，要熟悉与定时器有关的寄存器。总体而言，与定时器相关的寄存器主要有以下几种。

- 定时器控制寄存器 TCON

由于各个定时器的工作原理相似，下面以定时器 0 为例进行讲解。在定时器控制寄存器 TCON 中，位[3:0]用于控制定时器 0，其含义如表 8-1 所示。

表 8-1 定时器控制寄存器 TCON

位	功能简述	描述	
0	开启/停止	0：停止定时器	1：开启定时器
1	手动更新	0：未使用	1：TCMPBO 和 TCNTBO 中的值会加载到寄存器 TCMPO 和 TCNTO 中
2	输出控制	0：当 TCMPO=TCNT0 时，TOUTO 引脚输出不翻转	1：当 TCMPO=TCNTO 时 TOUTO 引脚输出翻转
3	自动加载	0：不自动加载	1：当 TCNTO 的值减到 0 时，TCMPBO 中的值会加载到寄存器 TCMPO 和 TCNTO 中

- 定时器比较值、计数值缓存寄存器 TCMPBn、TCNTBn

这两个寄存器用于存储定时器的比较值和计数初始值。

- 定时器比较值、计数值寄存器 TCMPn、TCNTn

这两个寄存器是定时器内部寄存器，用户无须对其进行写操作。

- 定时器观察值寄存器 TCNTOn

在定时器减 1 计数过程中，TCNTn 的值可以通过读取 TCNTOn 寄存器得到。

- 定时器配置寄存器 TCFG0、TCFG1

前文讲到，PCLK 经过两级分频器，输出频率作为定时器的工作频率，如图 8-9 所示。因此，必然会存在寄存器来控制分频系数。

- ◆ 定时器配置寄存器 TCFG0 用于控制第 1 级分频器的分频系数，分频器输出频率为：PCLK/(prescaler value +1)，其中 prescaler value=0~255。
- ◆ 定时器配置寄存器 TCFG1 用于控制多路开关。

定时器的输入时钟=PCLK/(prescaler value +1) /(divider value)。

其中：

```
prescaler value=0~255
divider value=2,4,8,16
```

从图 8-11 可以看到，定时器 0、1 公用一个第 1 级分频器，第 1 级分频器的分频系数由 TCFG0 的位[7:0]控制；定时器 2、3、4 公用另一个第 1 级分频器，该分频器的分频系数由 TCFG0 的位[15:8]控制。同时，从图 8-11 可以看到，第 2 级分频器的分频系数是确定的，只有 5 种类型：2 分频、4 分频、8 分频、16 分频和外接时钟 TCLKn（n=0 或 1），定时器配置寄存器 TCFG1 用于控制多路选择开关，选择具体使用这 5 种频率中的哪一种。以定时器 0 为例，TCFG1 的位[3:0]用于控制定时器 0，其含义如表 8-2 所示。

表 8-2　定时器配置寄存器 TCFG1

TCFG1	位	描　述	
多路开关 0	[3：0]	为定时器 0 选择多路开关的输入	
		0b0 000	1/2
		0b0 001	1/4
		0b0 010	1/8
		0b0 011	1/16
		0b01xx	外部时钟 TCLK0

例 1：结合上面的讲解可知，定时器 0 的输入时钟是经过 PCLK 分频得到的，如图 8-11 所示向读者展示了其产生过程。

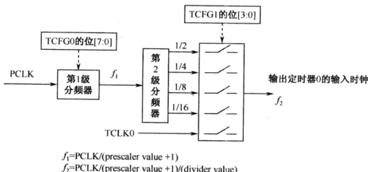


图 8-11　定时器 0 的输入时钟产生过程

例 2：设置适当的分频系数，使定时器 0 的输入时钟为 62.5 kHz。

因为 PCLK 为 50 MHz，则 50MHz/62.5 kHz = 800，即需要对 PCLK 进行 800 分频。所以，使第 1 级分频器的分频系数为 100，第 2 级的分频系数为 8 即可满足要求。最后，只需要将分频系数写入定时器控制寄存器中相应的位即可，代码如下：

```
1    rTCFG0   &=   ~(0xFF) ;
2    rTCFG0   |=   99 ;
3    rTCFG1   &=   ~(0xf) ;
4    rTCFG1   |=   0x02 ;
```

当向寄存器中的某几位写入数据时，请读者注意参考上面的代码。总体思路是，先将这几位清零，然后将数据写入这几位。

第 1 行，将 TCFG0 的低 8 位清零。

第 2 行，将分频系数−1 写入 TCFG0 的低 8 位，因为分频系数=prescaler value+1，即 prescaler+1= 100，所以，prescaler value = 99。

第 3 行，将 TCFG1 的低 4 位清零。

第 4 行，将 TCFG1 的低 4 位赋值为 0x02，即选择 8 分频输出。

8.2.3 定时器基础实验代码详解及测试

实验实现的功能：使用定时器 0 的定时功能，使 LED 每秒钟闪烁一次。

因为在启动代码阶段，已经对系统时钟进行了初始化，PCLK=50 MHz，定时器的输入频率由 PCLK 分频得到，如图 8-12 所示向读者展示了从晶振输入频率到定时器工作频率产生的整体过程。

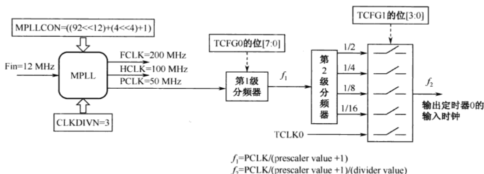


图 8-12　定时器 0 输入时钟一览

定时器工程的文件布局如图 8-13 所示。

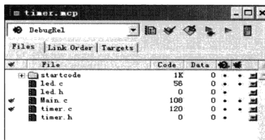


图 8-13　定时器工程的文件布局

定时器模块包含两个文件，即 timer.h 和 timer.c 文件。

timer.h 文件中声明了定时器 0 初始化函数 Timer0_Init()。

```
#ifndef _TIMER0_H_
#define _TIMER0_H_

void Timer0_Init(void) ;
```

```
#endif
```

timer.c 文件对定时器 0 初始化函数 Timer0_Init()进行了具体的实现。

```
void Timer0_Init(void)
  {
1    rTCFG0    &=  ~(0xFF) ;
2    rTCFG0    |=  99 ;
3    rTCFG1    &=  ~(0xf) ;
4    rTCFG1    |=  0x02 ;
5    rTCNTB0   =   62500 ;    //1s 中断一次
6    rTCON     |=  (1 << 1) ;
7    rTCON     =   0x09 ;
  }
```

在此实验中，定时器 0 的输入时钟频率为 62.5 kHz，即定时器每秒钟计数 62 500 次。因此，初始化时，定时器 0 初始值缓存寄存器中的值为 62 500，如第 5 行所示。

第 6 行，开启手动更新位，即当定时器开启后，TCNTB0 中的初始值会加载到内部寄存器 TCNT0 中。

第 7 行，关闭手动更新位，设置自动加载位，同时开启定时器，这样，TCNT0 进行减 1 计数，当 TCNT0 中的计数值减到 0 时，TCNTB0 中的数据会自动加载到 TCNT0 中并进行新一轮的减 1 计数。

用户主函数 Main.c 文件中的内容如下：

```
#include "led.h"
#include "timer.h"
int Main()
{
    int flag = 0 ;
    Led_Init() ;
    Timer0_Init() ;
    while(1)
    {
        if(rSRCPND & (1 << 10) )
        {
            flag = !flag ;
            rSRCPND |= (1 << 10) ; //清除定时器 0 中断标志位
        }
        if (1 == flag)
        {
            Led1_On() ;

        }
        else
```

```
            {
                Led1_Off();
            }

        }
        return 0;
    }
```

程序的基本原理如下。

程序开始进行了 LED 和定时器 0 的初始化，在定时器 0 初始化最后，打开了定时器 0，定时器 0 进行减 1 计数。

当 TCNT0 中的计数值减为 0 时，定时器 0 中断标志会置位。因此，在程序中可以通过不断地检测该位是否置位来判断 1s 定时是否到达。同时，TCNTB0 中的值会自动转入到 TCNT0 中，进行新一轮计数。

注意：虽然定时器 0 中断标志还没有讲解（在第 11 章讲解中断时，会系统地讲解定时器 0 中断的应用），但是，在此读者只需要了解以下三点。

第一，SRCPND 寄存器中的每一位代表一种类型的中断标志，当该位置 1 时，说明相应的中断发生了。

第二，定时器 0 中断标志位于 SRCPND 寄存器的第 10 位，当定时器 0 中的计数值减到 0 时，会触发定时器 0 中断标志，即 SRCPND 寄存器第 10 位会置 1。

第三，清除定时器 0 中断标志的方法是：向 SRCPND 寄存器的第 10 位写入 1 即可。

因此，rSRCPND & (1 << 10)的作用就是判断 SRCPND 寄存器的第 10 位是否为 1。如果为 1，说明 1s 时间到。然后，将 flag 取反，同时清除中断标志位，向 SRCPND 寄存器的第 10 位写 1 即可（将某一位置 1 时使用按位或指令）。最后，因为 flag 要么等于 1，要么等于 0，当 flag=1 时，点亮 LED；当 flag=0 时，熄灭 LED。

小技巧：判断 flag 是否为 1，可以使用 if(flag == 1)，但是，建议读者使用 if(1 == flag)，使用后者有以下优点：

当读者误写为 if(1 = flag)时，程序编译是不通过的，因为 flag 是个变量，1 是个常量，不能将变量赋值给常量，这就是所谓的语法错误，编译器可以发现语法错误。

如果使用前者，当读者在编程过程中不小心将 if(flag == 1)写为 if(flag = 1)后，编译阶段是不会发现的。因为编译器会误以为 flag =1 是个赋值语句，因此会通过编译，但是这与读者本来的意思是矛盾的，这就是所谓的逻辑错误，编译器是无法识别逻辑错误的。作为一个好的程序员，应尽量减少逻辑错误以大大提高软件开发的效率。

最后就是编译、链接生成.bin 格式的二进制文件，将其下载到 NAND FLASH 或者 NOR FALSH 中，启动开发板，此时 LED 已经闪烁起来了。

8.2.4 定时器扩展实验之 PWM 实验

上述内容主要是针对定时器的原理进行的实验，下面的实验向读者展示了定时器的脉冲宽度调制功能，即 PWM 功能。

从图 8-10 中可以清楚地看到，当 TCMP0=TCNT0 时，TOUT0 引脚电平会发生翻转，查询 S3C2440 数据手册可以得到，TOUT0 引脚对应的是 GPB0 引脚。此外，当 TCNT0 中的计数值减为 0 时，TOUT0 引脚电平再次发生翻转。因此，可以利用 TOUT0 引脚电平的两次翻转进行脉冲宽度调制，即 PWM。

在上述实验的基础上，修改 timer.c 中定时器 0 初始化函数。修改后，timer.c 文件内容如下：

```
void Timer0_Init(void)
{
1      rGPBCON &= ~(3 << 0) ;
2      rGPBCON |= 1 << 1 ;

3      rTCFG0 &= ~(0xFF) ;
4      rTCFG0 |= 99 ;
5      rTCFG1 &= ~(0xf) ;
6      rTCFG1 |= 0x02 ;
7      rTCNTB0 = 62 ;    //1s 中断一次
8      rTCMPB0 = rTCNTB0/2 ;
9      rTCON |=   (1 << 1)   ;
10     rTCON = 0x0D ;

}
```

第 1～2 行，配置 GPB0 引脚为 TOUT0 功能。

第 3～6 行，配置定时器的工作频率为 62 500 Hz。

第 7～8 行，设置 TCNTB0 初始值为 62，并且 TCMPB0 为 TCNTB0 的一半。

第 9～10 行，开启定时器，设置当 TCMP0=TCNT0 时，TOUT0 引脚电平发生翻转。

程序总体流程：

（1）开启定时器后，TCNTB0、TCMPB0 中的值分别装入 TCNT0 和 TCMP0 中，然后，TCNT0 从初值 62 开始减 1 计数。

（2）当 TCNT0=TCMP0=31 时，TOUT0 引脚电平发生翻转。

（3）TCNT0 继续减 1 计数，当 TCNT0 减为 0 时，TOUT0 引脚电平再次发生翻转，此时恰好产生一个方波。

（4）TCNTB0、TCMPB0 中的值被自动装入 TCNT0 和 TCMP0 中，TCNT0 继续从初值 62 开始减 1 计数。

由于此时定时器 0 的工作频率是 62 500 Hz，即每秒钟可以计数 62 500 次，而其初值为 62，所以所产生方波的频率为 62 500/62=1 008 Hz。又因为 TCMP0=TCNT0/2，所以产生的方波高电平持续时间和低电平持续时间各占一半。

如图 8-14 所示是用泰克示波器 DS1302 实际测试的结果，频率为 992.1 Hz，占空比约为 50%（由于测试环境和仪器的因素，测试频率存在一定的偏差）。

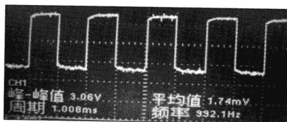


图 8-14　PWM 波形——TCM0P=TCNT0/2

将程序第 8 行修改为 rTCMPB0 = rTCNTB0/6，测试效果如图 8-15 所示。可见，输出方波的占空比发生了变化，这就是所谓的 PWM 功能。

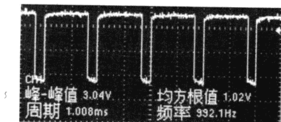


图 8-15　PWM 波形——TCMP0=TCNT0/6

8.3　本章小结

在本章中，重点对系统时钟电路和定时器进行了讲解，以理论为指导，配合了恰当的实验，向读者展示了启动代码中的系统时钟初始化以及定时器的具体应用。

第 9 章 存储器控制器

S3C2440 内存控制器提供了 CPU 访问外部设备所需的信号。

它具有以下特点：

- 支持小/大端（可以通过软件选择，默认情况下是小端格式）。
- 地址空间：128MB/BANK，共 8 个 BANK，总寻址空间为 1 GB。
- 除 BANK0 外，其他 7 个 BANK 的接口线宽可选择（8/16/32 位），BANK0 只支持 16/32 位两种。
- BANK0~5 可以外接 SROM、ROM，BANK6~7 可外接 SROM、ROM、SDRAM。
- BANK0~6 的起始地址是固定的，BANK7 的起始地址可以通过程序控制。
- BANK6~7 的大小也是可以通过程序控制的。
- 每个 BANK 的存储器访问周期可以通过程序控制。
- 总线周期可以通过外部的“Wait”信号延长。
- 当外接 SDRAM 时，支持自刷新模式和省电模式。

9.1 S3C2440 地址空间

虽然 S3C2440 是一款 32 位的 CPU，也就是说具有 32 位的地址总线和数据总线宽度，具有 32 位的寄存器，但实际上，S3C2440 的地址线只引出了 27 根，即 ADDR0~ADDR26，因此最大寻址空间为 2^{27}=128 MB。但是，S3C2440 处理器可以寻址最大 1 GB 的空间，这又是怎么做到的呢？

S3C2440 处理器将 1GB 的寻址空间分为 8 个区域，每个区域称为一个 BANK，即 1GB 的存储空间被分为了 BANK0~BANK7，每个 BANK 的最大空间为 128 MB，所以访问每个只需要 27 根地址线，此外又引出了 8 根片选信号线 nGCSx，用于区别不同的 BANK。

概括起来，它访问某一个物理地址的过程为：

（1）使该 BANK 的片选信号线 nGCSx 有效，即选择了该 BANK。

（2）用 27 根地址线在该 BANK 内寻址，进而找到相应的存储单元。

S3C2440 处理器内部地址空间分布如图 9-1 所示。左边对应的是 BANK0 没有外接 NAND FLASH 的情况，右边对应 BANK0 外接 NAND FLASH。在这种情况下，在 BANK0 的 0 地址处有一个 Boot Internal SRAM，大小为 4KB，在讲解从 NAND FLASH 启动时曾提到 S3C2440 处理器内部有一个 Stepping stone，大小是 4 KB，因此可以推断，当从 NAND

FLASH 启动时，可以将内部的 Stepping stone 理解为这个 Boot Internal SRAM。

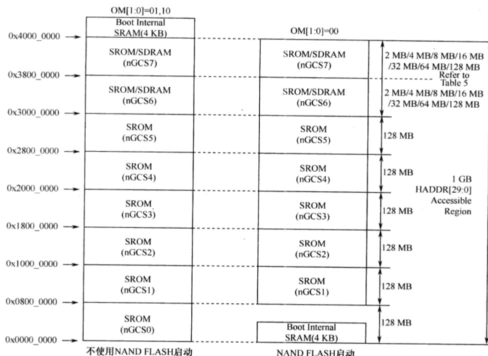


图 9-1　S3C2440 处理器内部地址空间分布

此外需要补充的是，理论上，S3C2440 处理器可以使用的物理地址空间可以达到 4 G。B，除去上述 1 GB 的外设地址空间外，还有一部分 CPU 内部使用的特殊功能寄存器地址空间，地址范围为：0x48 000 000~0x5FFFFFFF，其他的地址空间没有使用，如图 9-2 所示。

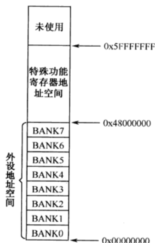


图 9-2　S3C2440 地址空间总图

对于 TQ2440 开发板，S3C2440 处理器外接了 NOR FLASH 和 SDRAM。其中，NOR FLASH 大小为 2 MB，接在 BANK0，SDRAM 大小为 64 MB，接在 BANK6 上，其地址范围如表 9-1 所示。

表 9-1　S3C2440 外设地址

外设名称	型号	BANKx	起始地址	结束地址	大小/MB	接口线宽
NOR FLASH	EN29LV160AB	BANK0	0x30 000 000	0x001FFFFF	2	16
SDRAM	MT48LC16M16A2	BANK6	0x30 000 000	0x33FFFFFF	64	32

9.2　操作实例：SDRAM 实例分析

同步动态随机存取内存（Synchronous Dynamic Random Access Memory，SDRAM）是一种具有同步接口的动态随机存取内存（Dynamic Random Access Memory，DRAM）。传统意义上的动态随机存取内存（DRAM）有一个异步接口，它可以随时响应控制输入的变化。SDRAM 有一个同步接口，在响应控制输入前会等待一个时钟信号，使得 RAM 和 CPU 能够共享一个时钟周期，以相同的速度同步工作，从而解决了 CPU 和 RAM 之间的速度不匹配问题，避免了在系统总线对异步 DRAM 进行操作时同步所需的额外等待时间，可加快数据的传输速度。

在嵌入式系统学习和开发过程中，SDRAM 的工作原理、控制时序及相关控制器的配置方法一直是初学者的一个难点。掌握 SDRAM 的基础知识对硬件设计、Bootloader 的编写、提高程序的执行效率都有一定的意义。

下面将以 Micron（美光）内存 MT48LC16M16A2-75D（TQ2440 开发板上用的是这种内存，不同时期生产的板子可能略有不同）为例，设计了 SDRAM 与 S3C2440A 的接口电路，详细分析了在初始化过程中各个参数的具体含义，旨在帮助初学者尽快掌握 SDRAM 的初始化工作。

9.2.1　SDRAM 工作原理

SDRAM 内部是一个存储阵列，如图 9-3 所示，由行（Row）、列（Column）和逻辑存储体（Logical Bank，简称 L-Bank）组成。内存芯片手册上经常用下述方式表示其内存容量。

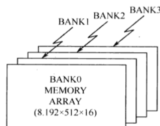


图 9-1　SDRAM 内部存储结构

每个 L-Bank 的容量×L-Bank 的位宽×L-Bank 的数目

本文所选的内存芯片 MT48LC16M16A2-75D 数据手册（如图 9-4 所示）上给出的指标为：

4 Meg × 16 × 4 banks

这表示该内存芯片有 4 个 L-Bank，每个 L-Bank 的位宽是 16 位，每个 L-Bank 的容量是 4 Mbit，该内存芯片的总容量为 256 Mbit，即 64 MB。

Micron

Synchronous DRAM

MT48LC64M4A2 - 16 Meg x 4 x 4 banks

MT48LC32M8A2 - 8 Meg x 8 x 4 banks

MT48LC16M16A2 - 4 Meg x 16 x 4 banks

For the latest data sheet,refer to Micron's Web site:www.micron.com

Features

- PC100-and PC133-compliant
- Fully synchronous;all signals registered on positive edge of system clock
- Internal pipelined operation;column address can be changed every clock cycle
- Internal banks for hiding row access/precharge
- Programmable burst lengths:1,2,4,8,or full page
- Auto precharge,includes concurrent auto precharge, and auto refresh modes
- Self refresh mode(not available on AT devices)
- Auto refresh
  - 64ms,8,192-cycle(commercial & industrial)
  - 16ms,8,192-cycle(automotive)
- LVTTL-compatible inputs and outputs
- Single+3.3V±0.3V power supply

图 9-4　MT48LC16M16A2-75D 数据手册

SDRAM 的基本读操作需要控制线和地址线相配合地发出一系列命令来完成。

（1）先发出 L-Bank 激活命令（ACTIVE），并锁存相应的 L-Bank 地址。

（2）然后指定一个行（Row），再指定一个列（Column）就可以准确地找到所需的单元格。

9.2.2　SDRAM 接口电路设计

在做 SDRAM 芯片扩展时，最主要的参考手册就是 S3C2440 数据手册，该手册给出的 SDRAM 配置实例如表 9-2 所示（本文只是摘取了手册中给出的部分内容）。

- Bank Size：表示 S3C2440 外接的 SDRAM 的总容量，其单位为 MByte。
- Bus Width：表示 S3C2440 与 SDRAM 接口位宽。
- Base Component：表示每片内存芯片的容量，单位为 Mbit。
- Memory Configuration：表示各内存芯片的构架和所需的内存芯片数目。
- Bank Address：表示内存芯片的选择信号。

表 9-2　S3C2440 SDRAM 配置实例

Bank size	Bus Width	Base component	Memory Configuration	Bank Address
64MByte	×32	128 Mbit	(4Meg×8×4 Banks)x4	**A[25:24]**
	×16	**256 Mbit**	(8Meg×8×4 Banks)x2	
	×32		**(4Meg×16×4 Banks)x2**	
	×8	512 Mbit	(16Meg×8×4 Banks)x1	

因为每块内存芯片的接口线宽是 16 位的，所以 TQ2440 开发板选择 2 片内存芯片 MT48LC16M16A2-75D 并联组成 32 位的位宽，与 CPU 的 32 根数据线相连，其接口电路如

图 9-5 所示。BANK6 的起始地址是 0x30000000，总容量为 64 MB，所以 SDRAM 的地址为：0x30000000～0x3FFFFFFF。

注意：在构建系统内存时，应根据系统的实际情况选择不同的内存芯片。在表 9-2 中给出了几种设计参考，加粗字体显示的配置情况为 TQ2440 开发板的实际内存配置。

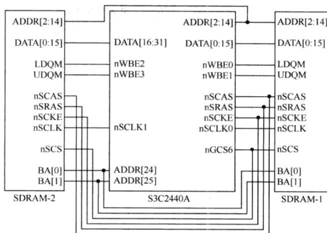


图 9-5　S3C2440A 与 SDRAM 接口电路

9.2.3　SDRAM 初始化过程详解

三星公司 S3C2440A 芯片内部存储器控制器总共有 8 个 BANK，其中 BANK6~BANK7 除了支持 ROM、SRAM 外，还支持 SDARM。存储器控制器共 13 个寄存器，BANK0~BANK5 只需要设置 BWSCON 和 BANKCONx（x：0~5），一般使用默认的 0x00 000 700 即可满足要求。

因为开发板上将 2 片内存芯片接在 BANK6 上，因此需要对 BANK6 设置 REFRESH、BANKSIZE、MRSRB6 3 个寄存器中相应的参数进行初始化。原因是：对 SDRAM 的访问需要遵循一定的 SDRAM 访问时序，而 S3C2440 处理器内部集成了 SDRAM 控制器，也就是上述 3 个寄存器，在开发过程中只需要根据相应内存芯片的数据手册中给出的时序参数（如图 9-6 所示）来设置内存控制器中相应的参数，内存控制器就可以产生相应的时序。

因此，本文重点讨论与 BANK6 有关的寄存器的初始化。

注意：在下文讨论中用到的参数都来自图 9-6。

- BANKCON6
  - MT([16:15])：用于控制外接 SRAM 或者 SDRAM、SRAM--0x02、SDRAM--0x03。
  - Trcd([3:2])：行地址到列地址的延时时间。该时间对应数据手册上的 tRCD(ACTIVE-to-READ or WRITE delay)，因为 TQ2440 开发板上 HCLK=100 MHz，因此每个时钟周期的时间为 1/100 M=10 ns，而芯片手册上 t_{RCD}=20 ns，所以需要两个时钟周期。由上述分析可知：Trcd 设为 0x0（2 clocks）即可。
  - SCAN([1:0])：SDRAM 列地址数。本文所选 SDRAM 的列地址数为 9，因此 SCAN 设为 0x01 即可。

Parameter		symbol	-75		Units	Notes
			Min	Max		
Access time from CLK (POSLTIVE EDGE)	CL=3	^{t}ac(3)	-	5.4	ns	27
	CL=2	^{t}ac(2)	-	6	ns	
Address hold time		^{t}AH	0.8	-	ns	
Address setup time		^{t}AS	1.5	-	ns	
CLK high-level width		^{t}CH	2.5	-	ns	
CLK low-level width		^{t}CL	2.5	-	ns	
Clock cycle time	CL=3	^{t}CK(3)	7.5	-	ns	23
	CL=2	^{t}CK(2)	1.0	-	ns	23
CKE hold time		^{t}CKH	0.8	-	ns	
CKE setup time		^{t}CKS	1.5	-	ns	37
CS#.RAS#.CAS#.WE#.DOM hold time		^{t}CMH	0.8	-	ns	
CS#.RAS#.CAS#.WE#.DOM setup time		^{t}CMS	1.5	-	ns	
Data-in hold time		^{t}DH	0.8	-	ns	
Data-in setup time		^{t}DS	1.5	-	ns	
Data-out High-Z time	CL=3	^{t}HZ(3)	-	5.4	ns	10
	CL=2	^{t}HZ(2)	-	6	ns	10
Data-out Low-Z time		^{t}LZ	1	-	ns	
Data-out hold time(load)		^{t}OH	3	-	ns	
Data-out hold time(no load)		^{t}OHN	1.8	-	ns	28
ACTIVE-to-PRECHARGE command		^{t}RAS	44	120 000	ns	
ACTIVE-to-ACTIVE command perlod		^{t}RC	66	-	ns	
ACTIVE-to-READ or WRITE delay		^{t}RCD	20	-	ns	
Refresh perlod（8,192 rows）		^{t}REF		64	ms	
Refresh perlod-Automotive(8,192rows)（8,192 rows）		^{t}REF$_{AT}$		16	ms	
Auto refresh perlod		^{t}RFC	66	-	ns	
PRECHARGE command perlod		^{t}RP	20	-	ns	
ACTIVE bank a to ACTIVE bank b command		^{t}RRD	15	-	ns	
Transition time		^{t}T	0.3	1.2	ns	7
Write recovery time		^{t}TWR	1CLK+7.5ns	-	ns	24
				-		
			15	-	ns	25
Exit SELF REFRESH to ACTIVE command		^{t}XSR	75	-	ns	20

图 9-6 SDRAM 芯片时序参数

- REFRESH
  - ◆ REFEN([23])：由于 SDRAM 利用其内部的电容来存储数据，而该存储单元存在漏电现象，为了保持每个存储单元数据的正确性，需要以一定的周期对其刷新。不同的内存芯片，刷新周期不同。对本文所用的内存芯片而言，数据手册指出其刷新周期是 64 ms，即存储体中电容的数据有效期上限是 64 ms，也就是说每一行刷新的循环周期是 64 ms。因此，此位设为 1，即启用 SDRAM 刷新功能。
  - ◆ TREFMD([22])：SDRAM 刷新模式控制位。通常，SDRAM 的刷新操作分为两种：自动刷新（Auto Refresh，AR）与自刷新（SelfRefresh，SR）。当 SDRAM 处于正常工作模式时，常选择自动刷新模式；当系统处于休眠状态时，选用自刷新模式。因此，此位设为 0。
  - ◆ Trp([21:20])：SDRAM 预充电时间。查询芯片数据手册得到 t_{RP}=20 ns，由于本系统 HCLK=100 MHz，时钟周期为 10 ns，所用只需要 2 个时钟周期即可。所以 Trp 设为 0（2 clocks）。
  - ◆ Tsrc([19:18])：SDRAM 半行周期时间。SDRAM 行周期时间满足 Trc=Tsrc+Trp，查询数据手册知：t_{RC}=66 ns，t_{RP}=20 ns，所以 Trc 为 7 clocks，Trp 为 2 clocks，可得 Tsrc 为 5 clocks，即 Tsrc=0x01。
  - ◆ Refresh Counter([10:0])：该值可用下述公式计算。

    （211−SDRAM 工作频率(MHz)×SDRAM 刷新周期(μs) +1）/HCLK

内存芯片手册上有这样一行：64 ms, 8,192-cycle refresh，即刷新 8192 行的时间是 64 ms，则刷新周期为 64 ms/8192=7.81 μs，因此 211+1−7.81×100=1 268=0x4F4。

- BANKSIZE

BK76MAP([2:0])：设置 BANK6/7 的大小。该位应当根据外接的 SDRAM 的实际大小选择相应的参数，本系统外接 64 MB 的 SDRAM，因此设为 0x01 即可。

- MRSRB6

在该寄存器中，能修改的位只有 CL([6:4])，这是 SDRAM 的一个时间参数，数据手册上 t_{CL}=2.5 ns，因此该位应设为 0x11。

9.2.4 回顾启动代码中的 SDRAM 初始化

首先，在 memcfg.inc 文件中对下列常量进行了定义。

```
1  ;Bank 6 parameter
2  B6_MT      EQU    0x3    ;SDRAM
3  B6_Trcd    EQU    0x0    ;2clk
4  B6_SCAN    EQU    0x1    ;9bit
5
6  ;Bank 7 parameter
7  B7_MT      EQU    0x3    ;SDRAM
8  B7_Trcd    EQU    0x0    ;2clk
9  B7_SCAN    EQU    0x1    ;9bit
```

```
10  ;REFRESH parameter
11  REFEN       EQU     0x1     ;Refresh enable
12  TREFMD      EQU     0x0     ;CBR(CAS before RAS)/Auto refresh
13  Trp         EQU     0x0     ;2clk
14  Tsrc        EQU     0x1     ;5clk    Trc= Trp(3)+Tsrc(5) = 8clock
15  REFCNT      EQU     1268    ;HCLK=100Mhz, (2048+1-7.81*100);
```

其次，在 init.s 文件中包含了 memcfg.inc 文件，然后就可以在 init.s 文件中使用上述变量了。

```
16  GET memcfg.inc
……
17  adrl        r0, SMRDATA         ;please cautionl!
18   ldmia      r0, {r1-r13}
19   ldr        r0, =BWSCON
20   stmia      r0, {r1-r13}
……
21  SMRDATA
22          DCD     0x22011000
23          DCD     0x00000700      ;GCS0
24          DCD     0x00000700      ;GCS1
25          DCD     0x00000700      ;GCS2
26          DCD     0x00000700      ;GCS3
27          DCD     0x00000700      ;GCS4
28          DCD     0x00000700      ;GCS5
29          DCD     ((B6_MT<<15)+(B6_Trcd<<2)+(B6_SCAN))        ;GCS6
30          DCD     ((B7_MT<<15)+(B7_Trcd<<2)+(B7_SCAN))        ;GCS7
31          DCD  ((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Tsrc<<18)+REFCNT)
32          DCD     0xB1
33          DCD     0x30            ;MRSR6 CL=3clk
34          DCD     0x30            ;MRSR7 CL=3clk
```

需要提醒读者注意，第 29~31 行，这是一种常见的利用 C 语言中移位运算定义常量的方法。到此，就完成了内存控制器的初始化。

9.3 本章小结

本章中对 S3C2440 处理器的存储器控制器进行了初步讲解，重点讲解了 SDRAM 的工作原理与初始化方法。在初学阶段，这些知识已经足够用，但是 ARM 的存储器控制器本身也是比较复杂的，在以后的学习过程中，希望读者参考其他书籍进行深入的学习。

第10章

通用异步收发器（UART）

S3C2440 通用异步收发器（UART）提供 3 个独立的异步串行 I/O，每个端口可以工作在中断或 DMA 模式下。也就是说，在 CPU 和 UART 之间传输数据时，UART 可以产生中断或 DMA 请求。UART 支持位速率高达 115.2 Kbps。通过外接时钟 UEXTCLK，UART 可以以更高的速率工作。

10.1 UART 概述

使用 UART 的最简单情况是只使用 3 根线：Tx 用于数据发送，Rx 用于数据接收，GND 是双方地线，提供通信双方的参考电平，如图 10-1 所示。

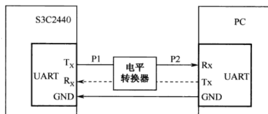


图 10-1　UART 接口原理图

注意： 电平转换器的作用是完成通信双方之间的电平转换。S3C2440 处理器输出电平是 CMOS 电平。对于 CMOS 电平，输入电压：$V_{IL}<0.3\times V_{cc}$，$V_{IH}>0.7\times V_{cc}$。输出电压：$V_{OL}<0.1\times V_{cc}$，$V_{OH}>0.9\times V_{cc}$，对于 S3C2440 而言，$V_{cc}=3.3$ V。

RS-232 最早是一种用在公用电话网的串行通信标准，传输距离一般不超过 15 m。

逻辑“0”：+3～+15 V。

逻辑“1”：−3～−15 V。

因此上述电平关系可通过图 10-2 来表示。

从图 10-2 中可以看出，RS-232 是用正负电压来表示逻辑状态，与 CMOS 以高低电平表示逻辑状态的规定不同。因此，为了能够同计算机接口或终端的 CMOS 器件连接，必须在 RS-232 与 CMOS 电路之间进行电平和逻辑关系的变换。

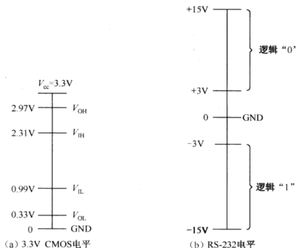


图 10-2　CMOS 电平和 RS-232 电平

10.2　S3C2440 处理器 UART 工作原理

S3C2440 的 UART 包括可编程的波特率，红外（IR）发射/接收，一个或两个停止位，5 位、6 位、7 位或 8 位的数据宽度和奇偶校验位。

每个 UART 包含一个波特率发生器、发送器、接收器和一个控制单元，如图 10-3 所示。波特率发生器的输入时钟有 3 种：PCLK、FCLK/ n、UEXTCLK（外部输入时钟）。

数据收发的基本原理如下。

UART 包含两种工作模式，FIFO 模式和非 FIFO 模式。

- FIFO 模式数据收发过程

发送数据：在发送数据之前，先将数据写入到发送 FIFO，然后数据从发送 FIFO 复制到发送移位寄存器，最后将数据从数据引脚（TxDn）移出。

接收数据：数据从 RXDn 引脚一位一位地接收到接收移位寄存器，然后数据从接收移位寄存器复制到接收 FIFO，最后，CPU 可以从接收 FIFO 中读取数据，如图 10-3 中黑色虚线所示。

- 非 FIFO 模式数据收发过程

发送数据：在发送数据之前，先将数据写入到发送保持寄存器，然后数据从发送保持寄存器复制到发送移位寄存器，最后将数据从数据引脚（TxDn）移出，如图 10-3 中黑色粗实线所示。

接收数据：数据从 RXDn 引脚一位一位地接收到接收移位寄存器，然后数据从接收移位寄存器复制到接收保持寄存器，最后，CPU 可以从接收保持寄存器中读取数据，如图 10-3 中黑色虚线所示。

注意：从图 10-3 中可以很容易看出发送 FIFO 和发送保持寄存器、接收 FIFO 和接收保持寄存器的关系。发送保持寄存器只是发送 FIFO 中的一个字节，接收保持寄存器只是

接收 FIFO 中的一个字节。其实，非 FIFO 模式可以理解为 FIFO 模式的一个特例，此时，FIFO 寄存器只有一个字节，而在 FIFO 模式时，FIFO 寄存器有 64 个字节。

作为入门级的学习，下面以非 FIFO 模式讲解。

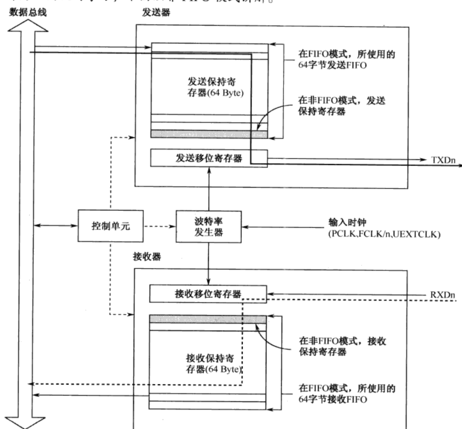


图 10-3　S3C2440 UART 内部原理图

10.3　引脚描述及相关寄存器

S3C2440 的引脚是复用的，可以通过编程将同一个引脚设置为不同的功能，UART 也不例外。前文讲到在最简单的情况下，UART 只需要 3 根线就可以实现通信功能，除去 GND 外，只有两根线：一根数据发送线 TXD，一根数据接收线 RXD。对于 UART0，查询数据手册可以知道，TXD0 与 GPH2 是复用的，RXD0 与 GPH3 是复用的。

因此，使用 UART0，首先应将 GPH2 设置为 TXD0 功能，将 GPH3 设置为 RXD0 功能：

```
rGPHCON  &=  ~((3 << 4) | (3 << 6));
rGPHCON  |=   (2 << 4) | (2 << 6);       //GPH2--TXD0;GPH3--RXD0
```

其次是初始化与 UART0 相关的寄存器。

注意： S3C2440 处理器串口具有很多功能。例如，支持 FIFO 模式、硬件流控、接收中断、接收超时、接收错误状态中断使能等功能。但是，初学者入门时不需要过多地了解这些功能，入门级的学习只需要实现如下功能：从计算机通过串口发送一个字符给 S3C2440，S3C2440 收到后通过串口发给计算机。

总体来讲，实现上述功能只需要初始化以下 6 个寄存器：ULCONn、UCONn、UBRDIVn、UTRSTATn、URXHn、UTXHn。在下文中，没有介绍上述寄存器中的某些无关位，有兴趣的读者可以参考 S3C2440 数据手册。

- ULCONn（UART LINE CONTROL REGISTER）

主要用于设置数据的长度、停止位和校验位信息，其格式如表 10-1 所示。

表 10-1　ULCONn 寄存器

ULCONn	位	描　述	初始状态
Reserved	[7]	-	0
红外模式	[6]	0：正常模式 1：红外发送/接收模式	0
校验位	[5：3]	0b0xx:无校验 0b100:奇校验 0b101:偶校验	000
停止位宽度	[2]	0：每帧数据有 1 个停止位 1：每帧数据有 2 个停止位	0
数据位宽度	[1：0]	0b00:5 位　0b01:6 位 0b10:7 位　0b11:8 位	00

- UCONn（UART CONTROL REGISTER）

主要用于设置数据发送和接收的模式，中断方式还是查询方式，其格式如表 10-2 所示。

表 10-2　UCONn 寄存器

UCONn	位	描　述
时钟选择	[11：10]	0b00/0b10:PCLK 0b01:　UEXTCLK 0b11:　FCLK/n
接收模式	[3：2]	0b00：关闭 0b01：中断方式或者查询方式
发送模式	[1：0]	0b00：关闭 0b01：中断方式或者查询方式

从表 10-2 中可以看到，UCONn 的第 10～11 位用于选择波特率发生器的输入时钟，读者可以结合图 10-3 理解。在本次实验中选择的是 PCLK。

- UBRDIVn （UART BAUD RATE DIVISOR REGISTER）

主要用于设置波特率。

UART 模块有 3 个 UART 波特率除数寄存器：UBRDIV0、UBRDIV1 和 UBRDIV2。根据所需的波特率和选定的时钟源，波特率除数寄存器（UBRDIVn）的值可以用如下公式

计算得到：

```
UBRDIVn = (int)( UART clock / ( baud rate × 16) ) –1
```

其中，UART clock 对应着图 10-3 中波特率发生器的 3 种输入时钟中的一个，baud rate 是用户所需要的波特率，最前面用了强制类型转换，将计算结果转换为整数存储在 UBRDIVn 中。

例：如果要求串口通信的波特率是 115 200，波特率发生器的输入时钟选择 PCLK=50 MHz，那么

UBRDIVn = (int)(50 000 000 / (115 200 × 16)) –1
= (int)(27.1) –1
= 27 –1
= 26

- UTRSTATn（UART TX/RX STATUS REGISTER）

该寄存器包含发送和接收是否完成的状态位，其格式如表 10-3 所示。

表 10-3　UTRSTATn 寄存器

UTRSTATn	位	描述
发送空	[2]	当发送缓冲器无有效数据且最后一字节数据被发送后，该位自动置 1 0：数据发送未完成 1：发送空
接收数据就绪	[0]	当接收缓冲器中接收到有效数据后，该位自动置 1 0：未接收到有效数据 1：接收到有效数据

例：可以使用 while(!(UTRSTAT0 & (1 << 2)))；语句来等待发送完成。

例：可以使用 while(!(UTRSTAT0 & (1 << 0)))；语句来查询是否接收到有效数据。

- URXHn（UART RECEIVE BUFFER REGISTER）

接收数据缓冲区寄存器，8 位数据长度，当接收到数据后，从 CPU 可以从该寄存器读取接收到的数据。

例：使用 UART 第 0 通道从 RXD 接收数据可以使用以下方法。

```
unsigned char c;
while(!(rUTRSTAT0 & RXD0READY)) ;    //等待接收完成
c = rURXH0;
```

- UTXHn（UART TRANSMIT BUFFER REGISTER）

发送数据缓冲区寄存器，8 位数据长度，当发送数据时，将要发送的数据写入该寄存器，即可自动发送。

例：使用 UART 第 0 通道发送数据可以使用如下方法。

```
rUTXH0 = c;
    while(!(rUTRSTAT0 & TXD0READY)) ;// 等待发送完毕
```

10.4 UART 基础实验

前面讲解了基础知识，下面结合硬件电路进行分析。

10.4.1 硬件电路分析

前文讲到，要想实现 PC 和 S3C2440 之间的 UART 通信，需要一个电平转换器，图 10-4（摘自“天嵌开发板电路图”）中的 SP3232EEN 就是一个电平转换芯片，完成所需的电平转换功能。

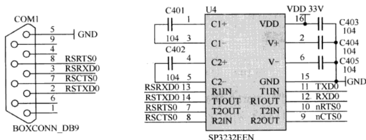


图 10-4　UART 硬件电路

初学者无须对该芯片的外接电容过多关注，只需按照芯片数据手册给出的典型应用电路接上合适的电容即可。图 10-5 是 SP3232E 电平转换芯片数据手册给出的典型应用电路。

此外需要注意的是，图 10-4 中还有 RSTRS0、RSCTS0、nTRS0 和 nCTS0 这 4 根线，主要用在硬件流控方面。在本实验中并没有使用硬件流控，初学阶段完全不必考虑。

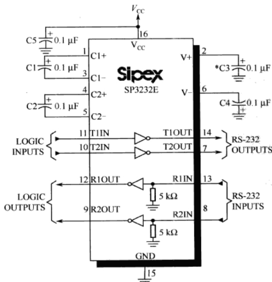


图 10-5　SP3232E 电平转换芯片数据手册给出的典型应用电路

10.4.2 程序设计及代码详解

实验实现的功能：PC 通过串口发送一个字符给 S3C2440，S3C2440 收到后通过串口发给 PC。

UART 工程的文件布局如图 10-6 所示。

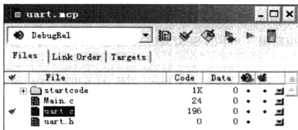


图 10-6 UART 工程文件布局图

UART 模块包含两个文件：uart.h 和 uart.c 文件。

- uart.h 文件中声明了 UART 初始化函数 Uart0_Init ()

```
#ifndef __UART_H__
#define __UART_H__

extern void Uart0_Init(unsigned int baudrate) ;
extern void putc(unsigned char c) ;
extern unsigned char getc(void) ;

#endif
```

- uart.c 文件对上述三个函数进行了具体的实现

```
#define PCLK 50000000                    //时钟源设为 PCLK
void Uart0_Init(unsigned int baudrate)
{
1    rGPHCON &= ~((3 << 4) | (3 << 6)) ;
2    rGPHCON |= ((2 << 4) | (2 << 6)) ;//GPH2--TXD[0];GPH3--RXD[0]
3    rGPHUP   = 0x00;
4    rULCON0 |= 0x03;                    //8 个数据位，1 个停止位
5    rUCON0   = 0x05;
6    rUBRDIV0 = (int) (PCLK / baudrate / 16)- 1;
7    rURXH0    = 0;
}
```

第 1～3 行，将 GPH2、GPH3 配置为 TXD、RXD 模式。

第 4 行，设置寄存器 ULCON0，设置数据发送格式为：8 个数据位、1 个停止位、无校验位。

第 5 行，发送模式和接收模式都使用查询方式。

第 6 行，设置波特率，其中波特率作为一个参数传递到该初始化函数。

第 7 行，将 URXH0 清零。

```
void putc(unsigned char c)
{

    rUTXH0 = c ;
    while(!(rUTRSTAT0 & (1 << 2))) ;//等待上个字符发送完毕
}
```

该函数可以发送一个字符。

首先将要发送的字符存入寄存器 UTXH0 中，然后等待发送完毕。发送完毕后，UTRSTAT0 寄存器的第 2 位会置 1，然后跳出 while 循环。

```
unsigned char getc(void)
{
    unsigned char c ;
    while(!(rUTRSTAT0 & (1 << 0))) ;
    c = rURXH0 ;
    return c ;
}
```

该函数实现接收一个字符，首先是等待接收完毕，接收完毕后，UTRSTAT0 第 0 位置 1，跳出 while 循环，将 URXH0 中的值读出即可。

用户主文件 Main.c 文件内容如下：

```
#include "uart.h"

int Main()
{
    unsigned char a ;
    Uart0_Init(115200) ;//初始化串口波特率为 115200

    while(1)
    {
        a = getc() ;
        putc(a) ;
    }
    return 0;
}
```

10.4.3 实例测试

编译、链接生成.bin 格式的二进制文件后下载到开发板，打开超级终端，波特率设为 11 5200，从键盘按下任一键，会发现超级终端上显示了用户按下的键，如图 10-7 所示，这

是说明 S3C2440 成功接收到了用户输入的数据，然后又将其发送到了 PC。

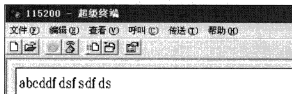


图 10-7 UART 基础实验测试

10.4.4 UART 基础实验分析

前文讲解了串口通信中的电平转换以及波特率等知识，但是并没有对其进行具体深入的讲解，下面结合泰克示波器 TDS2012 捕捉到的串口发送数据波形进行讲解，目的是教会读者如何利用已有的串口通信 UART 协议来分析示波器捕捉到的波形，进而得到上位机发送的数据。

本次实验的基本框图如图 10-8 所示。

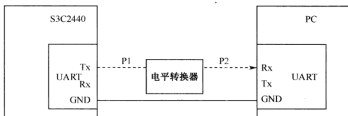


图 10-8 UART 发送数据波形分析辅助图

实验思路是：从 S3C2440 UART0 发送字符'a'，然后分别从 UART0 的 TXD0 引脚和经过 SP3232EEN 转换后的输出引脚来捕捉数据发送波形，即观测 P1 和 P2 点的波形。

- 在 UART 基础实验的基础上修改 Main.c 文件如下

```
#include "uart.h"

int Main()
{
    Uart0_Init(115200) ;//初始化串口波特率为 115200
    while(1)
    {
        putc('a');
    }
    return 0;
}
```

可见，串口通信的波特率为 115 200，在主循环中一直发送字符行数据'a'。

- S3C2440 UART0 TXD0 引脚波形

将程序下载到 S3C2440，上电后，将示波器探针接到 UART0 的数据发送引脚 TXD0，可以得到如图 10-9 所示的 P1 点的波形。

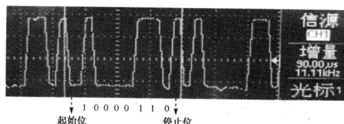


图 10-9　TXD0 引脚波形图

在图 10-9 中，可以找到起始位（电平从 1 变为 0），然后是发送的数据 10 000 110，最后是截止位（电平从 0 变为 1）。现在有两个问题：这个 10 000 110 代表什么呢？右边的增量 90.00 μs 是什么意思呢？

看一下程序里面的 putc('a')，是将字符型数据'a'发送，而'a'对应的 ASCII 码是 0x61，与其对应的二进制数为 01 100 001。程序中是发送的 01 100 001，但是为什么从示波器捕捉到的数据是 10 000 110 呢？仔细对比 01 100 001 和 10 000 110 可以发现，这两个数据就是高位和地位互换了。联系 UART 协议可知，串口发送数据时是先发送低位（LSB），然后发送高位（MSB）。

90.00 μs 又是什么意思呢？请读者看图 10-10。

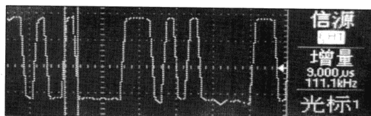


图 10-10　波特率辅助图

注意右边有个 9.000 μs。为什么是 9.000 μs 呢？在初始化时将串口的波特率设置为 115 200，也就是说每秒钟传输 115 200 个二进制位。传输一个二进制位的时间为：1s/115 200=8.68 μs。因为图 10-10 中两条竖线之间的宽带恰好是一个二进制位之间的时间（读者选择泰克示波器 TDS2012 光标测量时间功能），约为 9.000 μs，理论值是 8.68 μs，由于在测试过程中有点误差，因此测出来是 9.00 μs。到此可以知道：9.00 μs 就是一个二进制位的传输时间。

图 10-9 中连续的 4 个 0 又是怎么确定的呢？为什么不是 5 个或者 3 个 0 呢？知道了传输一位的时间，那么图 10-9 中连续的 4 个 0 就是通过总的持续时间（可以通过示波器测量得到）除以一位的传输时间得到的。

90.00 μs 恰好是 9.000 μs 的 10 倍，也就是传输 10 个二进制位的时间。在串口初始化阶段设置的串口通信的数据格式为：1 个起始位、8 个数据位、1 个停止位，恰好是 10 位，

这10位数据组合在一起叫做一帧数据。因此，在UART中，当串口通信的波特率为115 200时，传输一帧数据的时间为90.00 μs，理论上是86.8 μs，因此一般串口通信都存在一定的误差。

- 电平转换芯片SP3232EEN引脚波形

P2点的波形如图10-11所示。

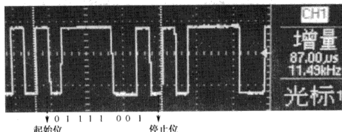


图10-11　SP3232EEN引脚波形

首先请读者注意SP3232EEN只是实现电平转换，不会修改发送的数据。在图10-11中，可以找到起始位（电平从1变为0），然后是发送的数据01 111 001，最后是截止位（电平从0变为1）。01 111 001又是什么意思呢？对比图10-9、图10-11不难发现，P1点发送的数据是10 000 110，经过转换器SP3232EEN后得到P2点的波形是01 111 001，恰好是每一位都进行了取反。请读者回顾图10-2，答案一目了然，RS-232电平如下。

逻辑“0”：+3～+15 V。

逻辑“1”：−3～−15 V。

因此，经过转换器后，电平进行了取反。

10.5　UART高级实验——可变参数函数在UART中的应用

在C语言中，使用printf()进行格式化输出非常方便，例如，printf("%d\n",a) 可以将a的值以十进制的格式输出，然后换行。printf()函数的原型为：int printf(const char *format, ...)，在函数参数中的...表示可变参数，即输入参数的个数不确定（例如，printf("%d\n",a)和printf("%d %d\n",a，b)都可以使函数进行正确的输出），这种输入参数不确定的函数就叫可变参数函数。在UART中能不能自己写一个类似于printf()的函数（例如，Uart0_Printf("%d",a)）呢？下面针对这一问题进行简要的讨论。

10.5.1　程序设计及代码详解

本实验基于前面的基础实验，只是修改了uart.h和uart.c文件。

uart.h文件中声明了UART初始化函数Uart0_Init ()和输出函数Uart0_Printf()。

```
#ifndef __UART_H__
#define __UART_H__
```

```
extern void Uart0_Init(unsigned int baudrate) ;
extern Uart0_Printf(const char *fmt,...) ;
#endif
```

uart.c 文件对上述函数进行了具体的实现。

```
#include "config.h"
#include "uart.h"
#include <stdarg.h>
#define PCLK 50000000//时钟源设为 PCLK

void Uart0_Init(unsigned int baudrate)
{
    rGPHCON &= ~((3 << 4) | (3 << 6)) ;//GPH2-GPH3 是 RX/TX
    rGPHCON |= ((2 << 4) | (2 << 6)) ;//GPH2--TXD[0];GPH3--RXD[0]

    rGPHUP = 0x00 ;

    rULCON0 |= 0x03 ;                    //8 个数据位，1 个停止位
    rUCON0 = 0x05 ;
    rUBRDIV0   = PCLK / baudrate / 16 - 1 ;
    rURXH0 = 0;
}
```

注意：使用可变参数函数时会用到 C 库里的宏，这时需要包含头文件 stdarg.h。Uart0_Init 函数没有变化，与基础实验中相同。

```
static void Uart0_SendByte(int data)
{
    if(data == '\n')
    {
        while(!(rUTRSTAT0 & (1 << 2)));//等待发送完成
      rUTXH0 = '\r';
    }
    while(!(rUTRSTAT0 & (1 << 2)));
     rUTXH0 = data   ;
}
```

Uart0_SendByte(int data)函数是向串口发送一个字节的数据。注意，在超级终端中使用的换行符是'\r'，因此当遇到'\n'时需要将其转换为'\r'，然后就是等待上一个字节数据发送成功后，将新发送的数据写入发送寄存器。

```
static void Uart0_SendString(char *pt)
{
    while(*pt)
```

```
        Uart0_SendByte(*pt++);
    }
```

Uart0_SendString(char *pt)函数调用 Uart0_SendByte(int data)函数发送数据。注意，这两个函数只是在 UART 模块中调用，在 Main.c 中并没有直接调用。因此，函数前面使用了 static 关键字，增强了函数的封装性。

```
void Uart0_Printf(const char *fmt,...)
{
    va_list ap;
    char string[50];

    va_start(ap,fmt);
    vsprintf(string,fmt,ap);
    va_end(ap);
    Uart0_SendString(string);
}
```

理解 Uart0_Printf(const char *fmt,...)函数需要了解下面的基础知识。

可变参数函数的参数列表分为两部分：固定参数和个数可变的可变参数。函数中至少有一个固定参数；可变参数由于个数不确定，声明时用"..."表示。

- va_list ap：定义了一个指向可变参数列表指针。
- va_start(ap, argN)：使参数列表指针 ap 指向函数参数列表中的第一个可变参数，argN 是最后一个固定参数。例如，当函数的声明是 void va_test(char a, char b, …)，则它的固定参数依次是 a，b，最后一个固定参数 argN 为 c，因此就是 a_start(ap, c)。
- va_end(ap)：清空参数列表，并置参数指针 ap 无效，该宏的作用是结束可变参数的获取。
- vsprintf()函数原型为 int vsprintf(char *string, char *format, va_list param)，其作用是将 param 按格式 format 写入字符串 string 中。

因此上述函数的基本流程是：

- 先开辟一块区域存储可变参数。
- 然后，调用 vsprintf()函数将可变参数按照指定的格式复制到缓冲区中。
- 最后调用 Uart0_SendString()函数将该缓冲区中的数据打印到串口中。

Main.c 文件内容如下：

```
#include "uart.h"

int Main()
{
    unsigned int a = 10 ;
    Uart0_Init(115200) ;
    while(1)
    {
```

```
        Uart0_Printf("Uart0_Printf test output is:%d\n",a) ;
    }
    return 0;
}
```

将串口初始化，波特率设为 115 200，然后主循环中一直调用 Uart0_Printf()函数，将 a 以十进制的格式输出到串口。

10.5.2 实例测试

编译、链接生成.bin 格式的二进制文件后下载到开发板，打开超级终端，波特率设为 115 200，超级终端上显示了程序执行的结果，如图 10-12 所示。可见，变量 a 以十进制的格式输出了。

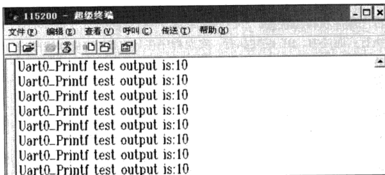


图 10-12 Uart0_Printf()函数输出

10.6 本章小结

本章中对 S3C2440 处理器的 UART 进行了初步讲解，目的是教会初学者简单应用 UART 的方法。在初始化阶段，涉及较多的寄存器，为了减小学习难度，摒弃了很多没有用到的位的设置。此外，主要是针对非 FIFO 模式进行的讲解，FIFO 模式较为复杂，并没有讲解。在收发数据过程中，采用的是查询的方式。在第 11 章中，会对串口的中断方式进行讲解。本章最后，扩展了可变参数函数在 UART 中的应用，对读者要求较高，初学阶段可以略过此部分。

第 11 章 中断控制系统

经过前面的学习，相信读者对 ARM 处理器的基本应用已经有了基本的认识，ARM 处理器程序的执行流程共分为 3 种。

- 正常执行：每执行一条 ARM 指令，程序计数器 PC 的值自动加 4。这一过程描述了应用程序顺序执行的状态。
- 跳转执行：通过 B、BL 跳转执行，实现程序在一定范围内的跳转执行。这一过程描述了 ARM 处理器程序执行过程中的过程调用。
- 中断处理：在应用程序执行过程中，发生中断后，ARM 处理器在执行完当前指令后，跳转到上述中断对应的中断处理程序处去执行，执行完中断处理程序后，再返回到发生中断的指令的下一条执行处接着执行。这描述了 ARM 处理器对异常中断的响应情况。

本章将对 ARM 中断系统进行详细的讲解，尽量将 ARM 处理器对中断处理的过程展示给读者，尽量使用图片向读者展示中断处理流程，力争将基本概念落实到具体的实验上，通过实验加深对具体理论的理解。

此外，在操作系统移植过程中可能会使用到软中断，软中断的概念不易理解，因此在本章中对软中断进行了讲解，同时给出了软中断的具体实验，希望通过一个简单的实验向读者展示出软中断的全貌。

注意：ARM 处理器异常处理情况是类似的，本章并没有对所有的情况进行讨论，只是对外部中断模式进行了深入讨论，希望读者可以举一反三，争取通过本章的学习，掌握其他几种异常处理。

11.1 S3C2440 中断系统概述

在 ARM 处理器运行过程中，需要与系统的各类外部设备进行通信，包括发出控制信息、读取外部设备的状态信息以及采集数据等。完成上述功能有以下两种实现方法。

- 查询方式：处理器不断地查询外部设备的状态，一般是每隔一段时间查询一次或者是始终在查询，这样做的缺点是实时性差、浪费处理器时间。
- 中断方式：当外设状态发生变化时发送一个信号给处理器，处理器通过内部的控制逻辑找到具体的外部设备，然后对其请求信息进行响应。在这种情况下，处理器的利用率大大提高，同时也提高了系统的实时性。

虽然本书针对 S3C2440 处理器进行讲解，但是 ARM 处理器对中断的处理流程基本符合下面的流程。

（1）中断发生后，中断控制器将中断请求发送给 CPU。

（2）CPU 将当前程序的运行环境（程序的运行环境一般是指程序执行过程中使用的寄存器以及程序的返回地址等信息）保存，调用相应的中断处理程序。

（3）在中断处理程序中，识别具体是哪个中断发生，并进行相应的处理。

（4）从中断处理程序返回，恢复被中断程序的运行环境，接着执行。

回顾第 1 章中提到的 ARM 处理器有 7 种工作模式，如表 11-1 所示。

表 11-1　ARM 处理器工作模式

处理器模式	说　明
用户模式（User）	程序正常执行的模式
快速中断模式（FIQ）	用于高速数据传输
外部中断模式（IRQ）	用于普通的中断处理
特权模式（Supervisor）	操作系统使用的一种保护模式
数据访问终止模式（Abort）	用于虚拟存储及存储保护
未定义指令终止模式（Undefined）	用于支持通过软件仿真硬件的协处理器
系统模式（Systerm）	用于运行特权级的操作系统任务

ARM 处理器的工作模式分为用户模式和特权模式，除用户模式外的其他 6 种工作模式为特权模式。此外，除用户模式和系统模式外的其他 5 种工作模式又称为异常模式。

ARM 处理器的工作模式可以通过软件改变，也可以通过外部中断或异常处理来改变处理器的工作模式。大多数的应用程序运行在用户模式下，当处理器运行在用户模式下时，某些被保护的系统资源是不能被访问的。

除了软件切换工作模式外，还有没有其他方法呢？通过异常中断的方式也可以进入相应的工作模式。例如，当 IRQ 中断发生时，处理器就进入外部中断模式（IRQ 模式）。

11.1.1　深入理解 CPU 的工作模式

ARM 寄存器共有 37 个寄存器，其中包括 31 个通用寄存器和 6 个状态寄存器，这些寄存器都是 32 位的。ARM 处理器的寄存器如图 11-1 所示。

ARM 处理器共有 7 种工作模式，每种工作模式都有对应的寄存器，在程序执行过程中，ARM 处理器肯定处于上述 7 种工作模式中的一种，在该工作模式中可见的寄存器（可见的寄存器即在该模式下可以访问的寄存器）主要有：15 个通用寄存器（R0～R14）、程序状态寄存器（CPSR 或者 SPSR）、程序计数器 R15（PC）。由图 11-1 可以看出，有些寄存器是各个模式公用的（也就是说，它们是同一个物理寄存器，因此当切换工作模式的时候，要保存这些寄存器的值），如 R0～R7，有些寄存器是各个模式独自拥有的物理寄存器（如图 11-1 中标黑色三角号的寄存器）。

现在有个问题：当进行工作模式的切换时，哪些寄存器的值需要保存呢？为什么快速中断模式比外部中断模式中断响应的速度要快？下面先详细讨论这个问题。

ARM State General Registers and Program Counter

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)

ARM State Program Status Registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

图 11-1 ARM 处理器的寄存器

假设 ARM 处理器处于用户模式（User 模式）执行程序，在程序执行过程中，发生了外部中断，则处理器进入外部中断模式（IRQ 模式）。从图 11-1 中可以看到，用户模式和 IRQ 模式的寄存器 R0～R12 是公用的，也就是说在进入 IRQ 模式之前，在用户模式下可能使用到了寄存器 R0～R12。如果在 IRQ 模式中也需要使用，那么寄存器 R0～R12 里面的值将被更改，在 IRQ 模式执行完中断处理程序，返回到用户模式后，寄存器 R0～R12 中的值被破坏了。因此，在 IRQ 模式中，使用寄存器 R0～R12 之前需要将其中的值保存，当从 IRQ 模式返回到用户模式时将原来的值恢复就可以避免上述问题。

从图 11-1 中还可以看到，用户模式的寄存器 R13～R14 和 IRQ 模式的寄存器 R13_irq、R14_irq 不是同一个寄存器（R13_irq、R14_irq 有个黑色小三角，说明这两个寄存器是独立的寄存器），也就是说在每种工作模式中，这两个寄存器是独立的。因此，当从用户模式切换到 IRQ 模式时不需要保存这两个寄存器的值。

此外，从图 11-1 中还可以注意到，用户模式和快速中断模式（FIQ 模式）公用的寄存器是 R0～R7，在 FIR 模式中，R8_fiq～R14_fiq 是独立的。因此，当从用户模式切换到快速中断模式（FIQ）时不需要保存这几个寄存器的值，只需要保存 R0～R7 的值即可。因此，发生快速中断时只需要保存 R0～R7，共 8 个寄存器。但是，当发生外部中断时需要保存 R0～R12 共 13 个寄存器，这里讲的保存寄存器的值是通过将其值入栈实现的，入栈是需要时间的，因此快速中断的响应时间要快一些。

11.1.2 中断控制器

S3C2440 中断控制器可以接收来自 60 个中断源的中断请求。对于 ARM 处理器而言，中断源很多，为了更好地处理各种中断，当中断发生时，中断请求并不是直接发送给 CPU，

而是发送给中断控制器（中断控制器需要用户进行初始化），中断控制器进行裁决后，选择出当前最需要处理的中断请求发送给 CPU，这样可以降低 CPU 的负担。

S3C2440 处理器的中断控制器结构如图 11-2 所示。

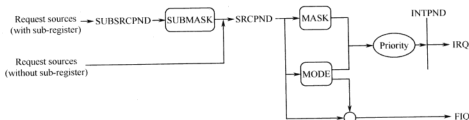


图 11-2　S3C2440 处理器的中断控制器结构

学习这部分内容，笔者没有直接按照 S3C2440 处理器数据手册进行翻译，而是尽可能地将问题简单化，针对一个具体的中断进行讲解，使读者对中断处理有个感性的认识，最后读者可以参阅 S3C2440 数据手册进行系统的学习。只要掌握了一个中断的处理流程，其他的中断都是类似的道理。

1．中断从哪里来

学习中断只需要弄清楚这样的问题：从哪里来，到哪里去，即中断是从哪里来的，中断发生后程序是如何执行的。

由图 11-2 可以看出，中断的来源是 Request sources（with sub-register）和 Request sources（without sub-register）。

- Request sources（with sub-register）主要处理以下类型的中断。例如，串口 UART，UART 中断包括发送中断、接收中断和因错误而产生的中断，但是这三个中断都属于一个大类，即都属于 UART 中断。如图 11-3 所示展示了这类子中断的处理流程。当 UART 发送中断产生时，SUBSRCPND 寄存器中响应的位会置 1，如果 INTSUBMSK 中不对其进行屏蔽，则 SRCPND 中的 UART 中断会置 1。

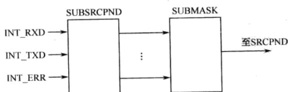


图 11-3　子中断的处理流程

技巧：到此为止不难发现，这里用了一个很简单的思想——分类汇总。

同种设备可能会对应几种中断，S3C2440 处理器采取的措施是：SRCPND 存储的是所有大类的中断，具体这一大类中断中哪种类型的子中断发生了，需要再查询 SUBSRCPND 寄存器才能得到。

例如，SRCPND 中的 UART 中断位置 1 了，说明发生了 UART 中断，但是具体是 UART 发送中断，还是 UART 接收中断，还是出错了，在该寄存器中无法确定具体哪种

类型的中断发生，因此还需要查询 SUBSRCPND 寄存器才能确定具体那种类型的中断发生了。

- Request sources（without sub-register）主要处理以下类型的中断。例如，外部中断 1，这种类型的中断只有一种情况，当外部中断的触发条件满足时就产生中断，这一类中断没有子类。

2．如何屏蔽中断请求信号

上面的讨论解决了中断“从哪里来”的问题。但是，请读者注意，并不是只要产生中断 CPU 就要对其进行响应，那么，如何使 CPU 不对某些中断进行响应呢？

最简单的情况是将该中断屏蔽掉。因此，这里就涉及一个中断屏蔽寄存器，只要向该寄存器中的某一位写 1，就可以将该位对应的中断屏蔽掉。

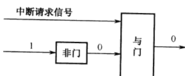


图 11-4　屏蔽中断信号

结合数字电路的知识来理解，如图 11-4 所示，假设某一个中断请求信号产生了，若此时中断屏蔽寄存器中该位的屏蔽信号为 1，则 1 经过非门，变为 0，中断请求信号与 0 相与，则输出为 0，即把中断请求信号屏蔽了。

中断屏蔽寄存器有两个（其实还有一个专门用于处理外部中断屏蔽的寄存器，对此将在外部中断实验部分讲解）：寄存器 INTSUBMSK 和寄存器 INTMSK。结合前文介绍的“分类汇总”的原理，屏蔽中断请求信号，需要屏蔽某一大类的中断时，就需要用到寄存器 INTMSK，屏蔽某一个子类的中断时就需要用到寄存器 INTSUBMSK。

例 1：屏蔽 UART0 中断。

寄存器 INTMSK 共 32 位，每一位对应一种类型的中断，当向该位写 1 时，就可以将该中断屏蔽，如图 11-5 所示。因此，要屏蔽 UART0 中断，只需要向寄存器 INTMSK 的第 28 位写 1 即可。代码如下：

```
#define   rINTMSK         (*(volatile unsigned *)0x4a000008)

rINTMSK |= 1 << 28 ;
```

例 2：只屏蔽 UART0 发送中断，但不屏蔽 UART0 接收中断。

由前面的分析可知，要实现上面的功能，需要打开 UART0 总中断，然后只屏蔽掉 UART0 发送中断即可。这时，需要用到子中断屏蔽寄存器 INTSUBMSK，该寄存器是 32 位的，但是高 17 位保留未用，只用到了第 15 位，每一位对应一种类型的子中断，如图 11-6 所示。

实现上述功能的代码如下：

```
#define rINTMSK         (*(volatile unsigned *)0x4a000008)
#define rINTSUBMSK     (*(volatile unsigned *)0x4a00001c)

rINTMSK        &=  ~(1 << 28) ;       //不需要屏蔽 UART0 总中断
rINTSUBMSK    |=   1 << 0 ;          //只需要屏蔽 UART0 发送子中断即可
```

INTMSK	Bit	Description	Initial State
INT_ADC	[31]	0 = Service available, 1 = Masked	1
INT_RTC	[30]	0 = Service available, 1 = Masked	1
INT_SPI1	[29]	0 = Service available, 1 = Masked	1
INT_UART0	[28]	0 = Service available, 1 = Masked	1
INT_IIC	[27]	0 = Service available, 1 = Masked	1
INT_USBH	[26]	0 = Service available, 1 = Masked	1
INT_USBD	[25]	0 = Service available, 1 = Masked	1
INT_NFCON	[24]	0 = Service available, 1 = Masked	1
INT_UART1	[23]	0 = Service available, 1 = Masked	1
INT_SPI0	[22]	0 = Service available, 1 = Masked	1
INT_SD1	[21]	0 = Service available, 1 = Masked	1
INT_DMA3	[20]	0 = Service available, 1 = Masked	1
INT_DMA2	[19]	0 = Service available, 1 = Masked	1
INT_DMA1	[18]	0 = Service available, 1 = Masked	1
INT_DMA0	[17]	0 = Service available, 1 = Masked	1
INT_LCD	[16]	0 = Service available, 1 = Masked	1
INT_UART2	[15]	0 = Service available, 1 = Masked	1
INT_TIMER4	[14]	0 = Service available, 1 = Masked	1
INT_TIMER3	[13]	0 = Service available, 1 = Masked	1
INT_TIMER2	[12]	0 = Service available, 1 = Masked	1
INT_TIMER1	[11]	0 = Service available, 1 = Masked	1
INT_TIMER0	[10]	0 = Service available, 1 = Masked	1
INT_WDT_AC97	[9]	0 = Service available, 1 = Masked	1
INT_TICK	[8]	0 = Service available, 1 = Masked	1
nBATT_FLT	[7]	0 = Service available, 1 = Masked	1
INT_CAM	[6]	0 = Service available, 1 = Masked	1
EINT8_23	[5]	0 = Service available, 1 = Masked	1
EINT4_7	[4]	0 = Service available, 1 = Masked	1
EINT3	[3]	0 = Service available, 1 = Masked	1
EINT2	[2]	0 = Service available, 1 = Masked	1
EINT1	[1]	0 = Service available, 1 = Masked	1
EINT0	[0]	0 = Service available, 1 = Masked	1

图 11-5　寄存器 INTMSK

INTSUBMSK	Bit	Description	Initial State
Reserved	[31:15]	Not used	0
INT_AC97	[14]	0 = Service available, 1 = Masked	1
INT_WDT	[13]	0 = Service available, 1 = Masked	1
INT_CAM_P	[12]	0 = Service available, 1 = Masked	1
INT_CAM_C	[11]	0 = Service available, 1 = Masked	1
INT_ADC_S	[10]	0 = Service available, 1 = Masked	1
INT_TC	[9]	0 = Service available, 1 = Masked	1
INT_ERR2	[8]	0 = Service available, 1 = Masked	1
INT_TXD2	[7]	0 = Service available, 1 = Masked	1
INT_RXD2	[6]	0 = Service available, 1 = Masked	1
INT_ERR1	[5]	0 = Service available, 1 = Masked	1
INT_TXD1	[4]	0 = Service available, 1 = Masked	1
INT_RXD1	[3]	0 = Service available, 1 = Masked	1
INT_ERR0	[2]	0 = Service available, 1 = Masked	1
INT_TXD0	[1]	0 = Service available, 1 = Masked	1
INT_RXD0	[0]	0 = Service available, 1 = Masked	1

图 11-6　寄存器 INTSUBMSK

3．中断模式

寄存器 INTMODE 是中断模式寄存器，S3C2440 处理器的中断分为一般的中断 IRQ 和快速中断 FIQ，INTMODE 寄存器的各位含义如图 11-7 所示。当 INTMODE 中某一位置 1 时，该位对应的中断被配置为快速中断模式。

关于 IRQ 和 FIQ 的讨论，在 11.1.1“深入理解 CPU 的工作模式”一节中已经进行了初步的讨论。本书主要是入门级的学习，因此关于快速中断不再进行讨论。INTMODE 不需要初始化也可以，此时所有的中断默认都是 IRQ 模式。

4．中断优先级

为了更好地响应各种类型的中断，S3C2440 处理器内部还有中断优先级寄存器，用于配置不同中断的优先级。

作为入门的学习，不需要考虑优先级的问题，采取默认的优先级即可。现在只有一个问题是主要的：中断到底是如何执行的？下面着重讲解这一过程。

INTMOD	Bit	Description	Initial State
INT_ADC	[31]	0=IRQ，　1=FIQ	0
INT_RTC	[30]	0=IRQ，　1=FIQ	0
INT_SPI1	[29]	0=IRQ，　1=FIQ	0
INT_UART0	[28]	0=IRQ，　1=FIQ	0
INT_IIC	[27]	0=IRQ，　1=FIQ	0
INT_USBH	[26]	0=IRQ，　1=FIQ	0
INT_USBD	[25]	0=IRQ，　1=FIQ	0
INT_NFCON	[24]	0=IRQ，　1=FIQ	0
INT_URRT1	[23]	0=IRQ，　1=FIQ	0
INT_SPI0	[22]	0=IRQ，　1=FIQ	0
INT_SDI	[21]	0=IRQ，　1=FIQ	0
INT_DMA3	[20]	0=IRQ，　1=FIQ	0
INT_DMA2	[19]	0=IRQ，　1=FIQ	0
INT_DMA1	[18]	0=IRQ，　1=FIQ	0
INT_DMA0	[17]	0=IRQ，　1=FIQ	0
INT_LCD	[16]	0=IRQ，　1=FIQ	0
INT_UART2	[15]	0=IRQ，　1=FIQ	0
INT_TIMER4	[14]	0=IRQ，　1=FIQ	0
INT_TIMER3	[13]	0=IRQ，　1=FIQ	0
INT_TIMER2	[12]	0=IRQ，　1=FIQ	0
INT_TIMER1	[11]	0=IRQ，　1=FIQ	0
INT_TIMER0	[10]	0=IRQ，　1=FIQ	0
ITN_WDT_AC97	[9]	0=IRQ，　1=FIQ	0
INT_TICK	[8]	0=IRQ，　1=FIQ	0
nBATT_FLT	[7]	0=IRQ，　1=FIQ	0
INT_CAM	[6]	0=IRQ，　1=FIQ	0
EINT8_23	[5]	0=IRQ，　1=FIQ	0
EINT4_7	[4]	0=IRQ，　1=FIQ	0
EINT3	[3]	0=IRQ，　1=FIQ	0
EINT2	[2]	0=IRQ，　1=FIQ	0
EINT1	[1]	0=IRQ，　1=FIQ	0
EINT0	[0]	0=IRQ，　1=FIQ	0

图 11-7　寄存器 INTMODE

11.2 外部中断实验

下面结合具体的实验，分析中断的响应过程，以及中断服务函数的编写。

本实验实现的功能：TQ2440 开发板上有 4 个按键，将这 4 个按键设置为外部中断方式，当按下 K1 时，LED1 亮；当按下 K2 时，LED2 亮；当按下 K3 时，LED3 亮；当按下 K4 时，LED4 亮。

11.2.1 硬件电路分析

TQ2440 开发板上共有 4 个 LED，其接口电路如图 11-8 所示。

LED 发光原理：只要 LED 中流过足够的电流，它就发光。现在就是不知道这个足够的电流是多少。其实在上述 LED 接口电路中，当 GPB5 输出低电平时，电阻和 LED 两端的总电压为 3.3 V，由欧姆定律的知识就可以很容易地判断流过 LED 的电流不会超过 3.3mA，因为 LED 也有内阻的。因此可以判断，这种 LED 发光时的电流是小于 3.3mA 的，一般 1～2mA 就会发光。

按键接口电路如图 11-9 所示。以 GPF1 为例，GPF1 通过一个 10 kΩ 上拉电阻接到电源，因此当按键没有按下时，GPF1 引脚电平为高电平，当按键按下时，引脚电平会变为低电平。因此，程序中就是通过对 GPF1 引脚的电平进行不断的检测，当引脚电平为低电平时说明按键被按下。

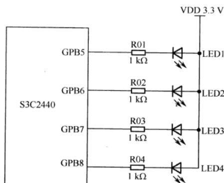


图 11-8　LED 接口电路

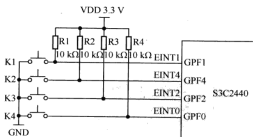


图 11-9　按键接口电路

11.2.2 程序分析

外部中断工程的文件布局如图 11-10 所示。

该工程由三个模块组成：按键模块、LED 模块和中断处理程序模块。按键模块主要包含 button.h 和 button.c 文件。LED 模块包含 ledflow.h 和 ledflow.c 文件。中断处理程序模块主要包含 interrupt.h、interrupt.c、isrservice.h 和 isrservice.c 文件。其中，interrupt.h 和 interrupt.c 文件主要包含中断初始化函数，isrservice.h 和 isrservice.c 文件主要包含中断响应函数，common.h

和 common.c 文件只是声明实现了一个简单的延时函数。下面将各个模块具体展开讲解。

按键模块包含两个文件：button.h 和 button.c 文件。

Eint.mcp

DebugRel

Files | Link Order | Targets

File	Code	Data
startcode	1K	0
Main.c	32	0
ledflow.c	56	0
ledflow.h	0	0
button.h	0	0
button.c	120	0
common.c	28	0
common.h	0	0
interrupt.c	96	0
interrupt.h	0	0
isrservice.c	328	0
isrservice.h	0	0

图 11-10　外部中断工程的文件布局

button.h 文件内容如下。

```
#ifndef __BUTTON_H__
#define __BUTTON_H__

extern void Key_Init() ;

#endif
```

声明了一个按键初始化函数 Key_Init()。

在 button.c 文件中对该函数进行了具体的实现：

```
#include "button.h"
#include "config.h"

#define   KEY1   (2 << 2)
#define   KEY2   (2 << 8)
#define   KEY3   (2 << 4)
#define   KEY4   (2 << 0)

void Key_Init()
{
1      rGPFCON   &=   ~((3 << 0) | (3 << 2) | (3 << 4) | (3 << 8)) ;
2      rGPFCON   |=    KEY1 | KEY2 | KEY3 | KEY4 ;
3      rGPFDAT   |=    (1 << 0) | (1 << 1) | (1 << 2 ) | (1 << 4) ;
}
```

第 1～2 行，因为按键此时作为外部中断输入，因此需要将按键对应的 GPIO 口配置外部中断模式，寄存器 GPFCON 的各位含义如图 11-11 所示。

对比图 11-9 和图 11-11 可得到按键与外部中断号的对应关系，如表 11-2 所示。

GPFCON	Bit	Description	
GPF7	[15:14]	00=Input 10=EINT[7]	01=Output 11=Reserved
GPF6	[13:12]	00=Input 10=EINT[6]	01=Output 11=Reserved
GPF5	[11:10]	00=Input 10=EINT[5]	01=Output 11=Reserved
GPF4	[9:8]	00=Input 10=EINT[4]	01=Output 11=Reserved
GPF3	[7:6]	00=Input 10=EINT[3]	01=Output 11=Reserved
GPF2	[5: 4]	00=Input 10=EINT[2]	01=Output 11=Reserved
GPF1	[3:2]	00=Input 10=EINT[1]	01=Output 11=Reserved
GPF0	[1:0]	00=Input 10=EINT[0]	01=Output 11=Reserved

图 11-11 寄存器 GPFCON

表 11-2 按键与外部中断号的对应关系

按 键	GPIO 引脚	外部中断号
K1	GPF1	外部中断 1 EINT1
K2	GPF4	外部中断 4 EINT4
K3	GPF2	外部中断 2 EINT2
K4	GPF0	外部中断 0 EINT0

第 3 行，因为在程序中检测按键是否按下，若按键按下，则相应的 GPIO 口电平变为低电平。因此，初始化阶段，使相应的 GPIO 口输出高电平。

LED 模块包含两个文件：ledflow.h 和 ledflow.c 文件。

ledflow.h 文件内容如下：

```
#ifndef __LEDFLOW_H__
#define __LEDFLOW_H__
#include "2440addr.h"

1    #define Led1_On()        {rGPBDAT &= (~(1 << 5));}
2    #define Led1_Off()       {rGPBDAT |= (1 << 5);}
3    #define Led2_On()        {rGPBDAT &= (~(1 << 6));}
4    #define Led2_Off()       {rGPBDAT |= (1 << 6);}
5    #define Led3_On()        {rGPBDAT &= (~(1 << 7));}
6    #define Led3_Off()       {rGPBDAT |= (1 << 7);}
7    #define Led4_On()        {rGPBDAT &= (~(1 << 8));}
8    #define Led4_Off()       {rGPBDAT |= (1 << 8);}

9    extern void Led_Init(void);
```

```
#endif
```

第 1～8 行，用宏定义的形式实现了 4 个 LED 的驱动。使用宏的形式实现这种简单的函数可以节省因函数调用而产生的开销，提高程序的运行效率。

第 9 行，声明了 LED 初始化函数 Led_Init()。

ledflow.c 文件内容如下：

```
#include "ledflow.h"
#include "2440addr.h"

void Led_Init(void)
{
	rGPBCON	&=	~((3 << 10) | (3 << 12) | (3 << 14) |(3 << 16)) ;
	rGPBCON	|=	((1<<10) | (1<<12) | (1<<14) | (1<<16)) ;
	rGPBUP	&=	~((1 << 5) | (1 << 6) | (1 << 7) || (1 << 8) ) ;
	rGPBDAT	|=	(1 << 5) | (1 << 6) | (1 << 7) | (1 << 8) ;
}
```

对于上述初始化函数，读者可以结合第 6 章进行理解。

下面重点分析中断相关的函数。中断处理程序模块主要包含 interrupt.h、interrupt.c、isrservice.h 和 isrservice.c 文件。

interrupt.h 文件内容如下：

```
#ifndef __INTERRUPT_H__
#define __INTERRUPT_H__

void Irq_Init(void) ;

#endif
```

在该文件中声明了中断初始化函数 Irq_Init()。

在讲解 interrupt.c 文件之前，需要了解知识点：初始化中断就是将这 4 个按键对应的中断屏蔽位置为无效。由图 11-5 可以看出，寄存器 INTMSK 中有单独的位来屏蔽外部中断 0～3，外部中断 4～7 是公用一个位来屏蔽的（为什么不是每个外部中断对应一个位呢？主要原因是外部中断太多了，因此需要另外一个寄存器 EINTMASK 来实现中断屏蔽）。具体屏蔽哪一位，需要由寄存器 EINTMASK 来确定。寄存器 EINTMASK 的各位含义如图 11-12 所示。

interrupt.c 文件内容如下：

```
#include "2440addr.h"
void Irq_Init(void)
{
1	rINTMSK		&=	~((1 << 0) | (1 << 1) | (1 << 2) | (1 << 4)) ;
2	rEINTMASK	&=	~(1 << 4) ;
}
```

EINTMASK	Bit	Description
EINT23	[23]	0=enable interrupt 1=masked
EINT22	[22]	0=enable interrupt 1=masked
EINT21	[21]	0=enable interrupt 1=masked
EINT20	[20]	0=enable interrupt 1=masked
EINT19	[19]	0=enable interrupt 1=masked
EINT18	[18]	0=enable interrupt 1=masked
EINT17	[17]	0=enable interrupt 1=masked
EINT16	[16]	0=enable interrupt 1=masked
EINT15	[15]	0=enable interrupt 1=masked
EINT14	[14]	0=enable interrupt 1=masked
EINT13	[13]	0=enable interrupt 1=masked
EINT12	[12]	0=enable interrupt 1=masked
EINT11	[11]	0=enable interrupt 1=masked
EINT10	[10]	0=enable interrupt 1=masked
EINT9	[9]	0=enable interrupt 1=masked
EINT8	[8]	0=enable interrupt 1=masked
EINT7	[7]	0=enable interrupt 1=masked
EINT6	[6]	0=enable interrupt 1=masked
EINT5	[5]	0=enable interrupt 1=masked
EINT4	[4]	0=enable interrupt 1=masked
Reserved	[3:0]	Reserved

图 11-12　寄存器 EINTMASK

第 1 行，将外部中断 0～3 的中断屏蔽位置为无效，只需要将其对应的位清零即可。注意，将外部中断 4 的中断屏蔽位置为无效需要执行下面两步：

（1）将寄存器 INTMSK 中 INT4_7 对应的屏蔽位置为无效。

（2）将寄存器 EINTMASK 中 INT4 对应的屏蔽位置为无效。

第 2 行就是将寄存器 EINTMASK 中 INT4 对应的屏蔽位置为无效。

注意：到此，外部中断的初始化工作结束。读者可能有这样的疑问：中断模式呢？中断优先级怎么配置呢？刚刚入门时可以不考虑这些，只要中断不被屏蔽，CPU 就可以收到中断信号，中断模式默认是 IRQ，中断优先级也有一个默认值。此外，具体的外部中断还可以选择触发方式，即高电平触发、低电平触发已经边沿触发等，这些由专门的寄存器（如外部中断控制寄存器 EXINTn）来设置，采取默认值即可，默认情况下是低电平触发。

下面的问题是：CPU 如何知道发生了中断呢？在处理器内部由专门的寄存器来记录哪个中断发生了。由图 11-2 可以看到，中断发生后，寄存器 SRCPND 中的相应位会置 1，然后，如果该中断不被屏蔽，则寄存器 INTPND 中的相应位也会被置 1。寄存器 SRCPND 的各位含义如图 11-13 所示，寄存器 INTPND 的各位含义如图 11-14 所示。

SRCPND	Bit	Description	Initial State
INT_ADC	[31]	0=Not requested, 1=Requested	0
INT_RTC	[30]	0=Not requested, 1=Requested	0
INT_SPI1	[29]	0=Not requested, 1=Requested	0
INT_UART0	[28]	0=Not requested, 1=Requested	0
INT_IIC	[27]	0=Not requested, 1=Requested	0
INT_USBH	[26]	0=Not requested, 1=Requested	0
INT_USBD	[25]	0=Not requested, 1=Requested	0
INT_NFCON	[24]	0=Not requested, 1=Requested	0
INT_UART1	[23]	0=Not requested, 1=Requested	0
INT_SPI0	[22]	0=Not requested, 1=Requested	0
INT_SD1	[21]	0=Not requested, 1=Requested	0
INT_DMA3	[20]	0=Not requested, 1=Requested	0
INT_DMA2	[19]	0=Not requested, 1=Requested	0
INT_DMA1	[18]	0=Not requested, 1=Requested	0
INT_DMA0	[17]	0=Not requested, 1=Requested	0
INT_LCD	[16]	0=Not requested, 1=Requested	0
INT_UART2	[15]	0=Not requested, 1=Requested	0
INT_TIMER4	[14]	0=Not requested, 1=Requested	0
INT_TIMER3	[13]	0=Not requested, 1=Requested	0
INT_TIMER2	[12]	0=Not requested, 1=Requested	0
INT_TIMER1	[11]	0=Not requested, 1=Requested	0
INT_TIMER0	[10]	0=Not requested, 1=Requested	0
INT_WDT_AC97	[9]	0=Not requested, 1=Requested	0
INT_TICK	[8]	0=Not requested, 1=Requested	0
nBATT_FLT	[7]	0=Not requested, 1=Requested	0
INT_CAM	[6]	0=Not requested, 1=Requested	0
EINT8_23	[5]	0=Not requested, 1=Requested	0
EINT4_7	[4]	0=Not requested, 1=Requested	0
EINT3	[3]	0=Not requested, 1=Requested	0
EINT2	[2]	0=Not requested, 1=Requested	0
EINT1	[1]	0=Not requested, 1=Requested	0
EINT0	[0]	0=Not requested, 1=Requested	0

图 11-13 寄存器 SRCPND

INTPND	Bit	Description	Initial State
INT_ADC	[31]	0=Not requested,　1=Requested	0
INT_RTC	[30]	0=Not requested,　1=Requested	0
INT_SPI1	[29]	0=Not requested,　1=Requested	0
INT_UART0	[28]	0=Not requested,　1=Requested	0
INT_IIC	[27]	0=Not requested,　1=Requested	0
INT_USBH	[26]	0=Not requested,　1=Requested	0
INT_USBD	[25]	0=Not requested,　1=Requested	0
INT_NFCON	[24]	0=Not requested,　1=Requested	0
INT_UART1	[23]	0=Not requested,　1=Requested	0
INT_SPI0	[22]	0=Not requested,　1=Requested	0
INT_SDI	[21]	0=Not requested,　1=Requested	0
INT_DMA3	[20]	0=Not requested,　1=Requested	0
INT_DMA2	[19]	0=Not requested,　1=Requested	0
INT_DMA1	[18]	0=Not requested,　1=Requested	0
INT_DMA0	[17]	0=Not requested,　1=Requested	0
INT_LCD	[16]	0=Not requested,　1=Requested	0
INT_UART2	[15]	0=Not requested,　1=Requested	0
INT_TIMER4	[14]	0=Not requested,　1=Requested	0
INT_TIMER3	[13]	0=Not requested,　1=Requested	0
INT_TIMER2	[12]	0=Not requested,　1=Requested	0
INT_TIMER1	[11]	0=Not requested,　1=Requested	0
INT_TIMER0	[10]	0=Not requested,　1=Requested	0
INT_WDT_AC97	[9]	0=Not requested,　1=Requested	0
INT_TICK	[8]	0=Not requested,　1=Requested	0
nBATT_FLT	[7]	0=Not requested,　1=Requested	0
INT_CAM	[6]	0=Not requested,　1=Requested	0
EINT8_23	[5]	0=Not requested,　1=Requested	0
EINT4_7	[4]	0=Not requested,　1=Requested	0
EINT3	[3]	0=Not requested,　1=Requested	0
EINT2	[2]	0=Not requested,　1=Requested	0
EINT1	[1]	0=Not requested,　1=Requested	0
EINT0	[0]	0=Not requested,　1=Requested	0

图 11-14　寄存器 INTPND

例如，当外部中断 0 发生时，寄存器 SRCPND 的第 0 位置 1，在初始化阶段，该中断请求并没有被屏蔽，因此寄存器 INTPND 的第 0 位也置 1。

注意：在寄存器 SRCPND 和 INTPND 中，外部中断 4～7（EINT4_7）是公用一位的。具体确定是哪一个中断发生时，还需要借助寄存器 EINTPEND，寄存器 EINTPEND 的各位含义如图 11-15 所示。

EINTPEND	Bit	Description	Reset Value
EINT23	[23]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT22	[22]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT20	[20]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT19	[19]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT18	[18]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT17	[17]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT16	[16]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT15	[15]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT14	[14]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT13	[13]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT12	[12]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT11	[11]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT10	[10]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT9	[9]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT8	[8]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT7	[7]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT6	[6]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT5	[5]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
EINT4	[4]	It is cleard by writing "1" 0=Not occur 1=Occur interrupt	0
Reserved	[3:0]	Reserved	0000

图 11-15 寄存器 EINTPEND

例如，当外部中断 4 发生时，寄存器 SRCPND 和 INTPND 的第 4 位置 1，同时寄存器 EINTPEND 的第 4 位也置 1，这时就可以确定是外部中断 4 发生了。又如，当外部中断 6 发生时，寄存器 SRCPND 和 INTPND 的第 4 位也会置 1，但此时寄存器 EINTPEND 的第 6 位会置 1，因此这样就可以进一步确定是外部中断 6 发生了。

最后的问题是：执行完中断响应函数后，如何清除中断呢？只需要向寄存器 SRCPND 和 INTPND 的相应位写 1 即可清除中断标志。对于外部中断 4～23，还需要清除寄存器

EINTPEND 中的相应位，也是向该位写 1 即可清除中断标志。

注意：需要先清除 SRCPND，然后再清除 INTPND。

例 1：清除外部中断 0 中断标志。

```
rSRCPND     |= 1 << 0 ;
rINTPND     |= 1 <<0 ;
```

例 2：清除外部中断 4 中断标志。

```
rSRCPND     |= 1 << 4 ;
rINTPND     |= 1 << 4 ;
rEINTPEND   |= 1 << 4 ;
```

对于 IRQ 模式的中断，S3C2440 处理器还提供了一个寄存器 INTOFFSET 用来标志寄存器 INTPND 的哪种类型中断发生了。寄存器 INTOFFSET 的各位含义如图 11-16 所示，当清除寄存器 SRCPND 和寄存器 INTPND 中相应的中断标志位后，寄存器 INTOFFSET 的值自动清零。

INT Source	The OFFSET value	INT Source	The OFFSET value
INT_ADC	31	INT_UART2	15
INT_RIC	30	INT_TIMER4	14
INT_SPI1	29	INT_TIMER3	13
INT_UART0	28	INT_TIMER2	12
INT_IIC	27	INT_TIMER1	11
INT_USBH	26	INT_TIMER0	10
INT_USBD	25	INT_WDT_AC97	9
INT_NFCON	24	INT_TICK	8
INT_UART1	23	nBATT_FLT	7
INT_SPI0	22	INT_CAM	6
INT_SDI	21	EINT8_23	5
INT_DMA3	20	EINT4_7	4
INT_DMA2	19	EINT3	3
INT_DMA1	18	EINT2	2
INT_DMA0	17	EINT1	1
INT_LCD	16	EINT0	0
OTE：FIQ mode interrupt does not affect the INTOFFSET register as the register is available only for IRQ mode interrupt.（注意：FIQ 中断不影响 INTOFFSET 寄存器的值。）			

图 11-16　寄存器 INTOFFSET

例如，若外部中断 0 发生且没有被屏蔽，则寄存器 INTOFFSET 的值为 0；若定时器 0 中断发生且没有被屏蔽，则寄存器 INTOFFSET 的值为 10。

经过前面的讲解，基本弄清除了下面几个问题：如何对外部中断进行初始化？CPU 如何检测哪个中断发生了？下面需要解决的问题是：CPU 检测到中断后要调用相应的中断处理函数，如何安装中断处理函数呢？

```
1    #define _ISR_STARTADDRESS        0x33ffff00
2    #define pISR_EINT0          (*(unsigned *)(_ISR_STARTADDRESS+0x20))
```

```
3    pISR_EINT0 = (unsigned int)Eint0_Isr ;

void __irq Eint0_Isr(void)
{
4    Led1_On();Delay1s() ;Led1_Off() ;
5    rSRCPND |= 1 << 0 ;//清除中断标志
6    rINTPND |= 1 << 0 ;
}
```

第 1～2 行，定义了一个指向内存地址 0x33ffff20（0x33ffff00+0x20）的指针。这里的 0x33ffff20 代表什么意思呢？回顾第 7 章的启动代码第 198～199 行之间有个注释：@0x33FF_FF20，因此这里的 0x33ffff20 就是内存中第 2 级中断向量的首地址。

第 3 行，将外部中断 0 响应函数的首地址加载到内存地址 0x33ffff20 处。

下面是用__irq 关键字声明的外部中断 0 中断处理函数。

第 4 行，使 LED1 闪烁一次。

第 5～6 行，清除外部中断 0 的中断标志，可以看到，先清除寄存器 SRCPND 中的相应位，然后清楚寄存器 INTPND 中的相应位。

注意：这里介绍得比较笼统（主要原因是，在此处主要是想突出安装中断处理程序的过程），读者可以结合 11.2.4 节进行学习。

前面只是介绍基本原理，下面介绍 isrservice.h 文件和 isrservice.c 文件。

isrservice.h 文件内容如下：

```
#ifndef __ISRSERVICE_h__
#define __ISRSERVICE_h__

void Isr_Init(void) ;
void __irq Eint0_Isr(void) ;
void __irq Eint1_Isr(void) ;
void __irq Eint2_Isr(void) ;
void __irq Eint4_7_Isr(void) ;
void __irq Timer0_Isr(void) ;

#endif
```

该文件主要声明了外部中断处理函数。

isrservice.c 文件内容如下：

```
#include"config.h"
#include"isrservice.h"
#include"ledflow.h"

void Isr_Init(void)
{
```

```
    pISR_EINT0 = (unsigned int)Eint0_Isr ;
    pISR_EINT1 = (unsigned int)Eint1_Isr ;
    pISR_EINT2 = (unsigned int)Eint2_Isr ;
    pISR_EINT4_7 = (unsigned int)Eint4_7_Isr ;
}
```

在此函数中实现了将不同的中断处理函数的入口地址加载到第 2 级中断向量表的相应地址处。

```
void __irq Eint0_Isr(void)
{
    Led1_On();Delay1s() ;Led1_Off() ;

    rSRCPND |= 1 << 0 ;
    rINTPND |= 1 << 0 ;

}

void __irq Eint1_Isr(void)
{
    Led2_On();Delay1s() ;Led2_Off() ;

    rSRCPND |= 1 << 1 ;
    rINTPND |= 1 << 1 ;
}

void __irq Eint2_Isr(void)
{
    Led3_On();Delay1s() ;Led3_Off() ;

    rSRCPND |= 1 << 2 ;
    rINTPND |= 1 << 2 ;
}

void __irq Eint4_7_Isr(void)
{
    unsigned long val ;
    val = rEINTPEND ;
    if(val & (1 << 4) )
    {
        rEINTPEND |= 1 << 4 ;
        Led4_On();Delay1s() ;Led4_Off() ;
    }
```

```
        rSRCPND     |= 1 << 4 ;
        rINTPND     |= 1 << 4 ;

    }
```

需要注意的是，对于外部中断 4 的处理需要进一步查询寄存器 EINTPEND 确定是哪一个外部中断发生了，最后清除中断标志时也需要将寄存器 EINTPEND 的相应位清除。

Main.c 文件内容如下：

```
#include "ledflow.h"
#include "button.h"
#include "isrservice.h"
void IO_Init() ;
int Main()
{
    IO_Init() ;
    while(1)
    {
        ;
    }
    return 0;
}

void IO_Init()
{
    Led_Init() ;
    Key_Init() ;
    Isr_Init() ;
    Irq_Init() ;
}
```

Main.c 文件只是将上述几个文件定义的函数进行了调用，然后执行无限循环，当有中断发生时，调用相应的中断处理函数即可。

到此为止，按照传统的思路，已经讲完了中断处理的全部流程。读者可以将程序编译，然后下载到开发板的 NAND FLASH。系统启动后，按下不同的键，会发现相应的 LED 灯已经点亮了，很多教程也是这么写的。但是，作为初学者，尤其是刚刚接触 ARM 的初学者，学习到这里，对 ARM 的中断处理流程依然是一头雾水……

编者认为，上述讲解仅仅是看到了 ARM 处理器中断处理的冰山一角，仅仅是一点点皮毛而已，因为它并没有触及 ARM 处理器中断处理的灵魂与精华。毕竟，下面的问题还没有解决（或者说，下面这些问题经常会使初学者迷茫与不解）。

- 当中断发生后，程序是如何跳转到中断处理函数的呢？
- 执行完中断处理函数后，如何返回到原来被打断的地方接着执行呢？
- ARM 处理器的流水线结构对中断返回地址的计算有什么影响呢？

- ARM 7 处理器是 3 级流水线结构，ARM 9 处理器是 5 级流水线结构，为什么中断返回地址的计算会相同呢？
- 前文讲解到，ARM 处理器有 7 种工作模式，发生中断后，处理器进入什么工作模式呢？
- 发生中断后，哪些事情是 ARM 处理器自动完成的呢？哪些事情是需要编程实现的呢？

11.2.3 中断执行流程详解

毕竟，中断处理才是学习 ARM 的难点所在，难点也往往是重点。下面针对上述问题进行全面讲解，力图尽量详细地向读者展现出 ARM 中断处理的全貌。

在系统启动后，系统进入管理模式（SVC 模式，这是 ARM 处理器上电后默认的工作模式），在发生中断后，ARM 处理器会自动切换到外部中断模式（IRQ 模式）（这是因为本文实验中将外部中断 0 设置为 IRQ 模式，如果设为快速中断模式，则发生中断后，处理器工作模式切换到快速中断模式，即 FIQ 模式）。前文讲解到，每种工作模式都对应一组可以访问的寄存器，如图 11-17 所示。

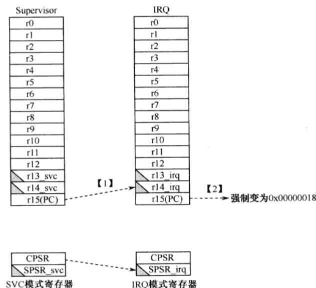


图 11-17 SVC 模式和 IRQ 模式寄存器

由图 11-17 可见，SVC 模式和 IRQ 模式寄存器的 r0～r12 是公用的，当前程序状态寄存器 CPSR 也是公用的。下面解释发生中断后，ARM 处理器做了哪些事情，还有哪些事情需要用户编程来实现。

发生中断后，ARM 处理器需要执行完当前指令，然后自动完成以下事情（如图 11-17 中虚线所示）：

- 将当前程序状态寄存器 CPSR 保存到 IRQ 模式下的备份程序状态寄存器 SPSR_irq

中（执行中断返回时，其逆过程不能自动完成）。

- 将程序计数器 PC 值减 4(这个地方要注意)存放到 IRQ 模式下的链接寄存器 R14_irq 中，即 R14_irq=PC-4。
- 最后将 PC 值强制设为 0x00000018（这正是异常中断向量表中 IRQ 的入口地址）。

注意：发生中断后，ARM 处理器需要执行完当前指令，然后再自动完成上述事情。因此，中断返回后执行当前指令的下一条指令。

从上面的过程可以得出，只要是 ARM 处理器没有自动完成的事情，都需要用户编程去实现。因此，程序员需要做的事情是：

- 根据需要，有选择地保存寄存器 r0～r12 的值（一般是将上述寄存器的值入栈保护）。
- 上面提到，中断发生后，ARM 处理器自动将程序计数器 PC 的值设为 0x00000018。因此，用户需要将中断处理函数放在这个地址处，一般是放置一条跳转指令，然后就可以找到中断处理函数。
- 计算程序的返回地址，因为 ARM9 处理器是 5 级流水线，但是程序的执行阶段是处在流水线的第三级。因此，程序寄存器 PC 始终指向当前正在执行指令的下两条指令处（初学者对此可以不必关心，只需要了解：PC 始终指向当前正在执行指令的下两条指令处）。

前文讲到需要计算程序的返回地址，那么如何计算呢？下面重点分析中断返回地址是如何计算出来的。

前文讲到 PC 始终指向当前正在执行指令的下两条指令处，如图 11-18 所示，因此，当前正在执行的指令地址就是 PC-8（ARM 指令是 4 字节对齐的，即每条指令占 4 个字节）。

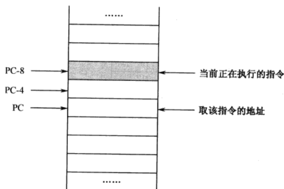


图 11-18　发生中断的指令

假设此时发生了中断，则 ARM 处理器并不是马上执行中断处理程序，而是先将当前指令执行完。注意，只要执行完该指令，PC 的值也已经更新了，即 PC 指向了下一条指令的地址处，如图 11-19 所示，然后自动将当前 PC-4 值保存到 IRQ 模式下链接寄存器 R14_irq 中。

这个地方需要注意，执行完中断处理程序后，需要返回的地址是发生中断的指令（注意，此时 PC 值已经更新了，如图 11-19 中灰色所示）的下一条指令的地址，即 PC-8 处，然后将 PC 值设为 0x00000018，接着就是访问异常中断向量表，查找到中断处理函数并执行，最后执行中断返回。

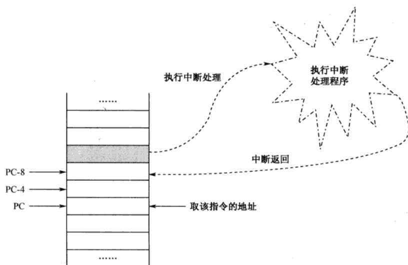


图 11-19 中断返回地址计算

中断返回执行的操作是：

（1）将被中断程序的处理器状态（寄存器的内容）恢复，即从 IRQ 模式恢复到 SVC 模式。

（2）返回到发生异常中断的指令的下一条指令处执行。

可用如下代码实现：

```
subs   pc, lr,    #4
```

该指令实现的功能：因为是从 IRQ 模式返回到 SVC 模式，所以上述指令是在 IRQ 模式下执行的，则 lr 寄存器指的是 IRQ 模式下的寄存器 r14_irq，将其值减去 4 恰好是中断返回的地址，同时该指令还自动将 SPSR_irq 的值复制到 CPSR 寄存器中，执行过程如图 11-20 所示。

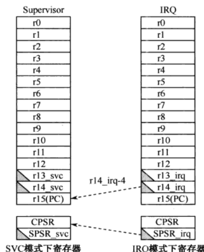


图 11-20 中断返回

此外，也可以利用入栈和出栈指令实现：

```
subs    lr,  lr,  #4
stmfd   sp!,      {r0-r12,lr}
......
ldmfd   sp!,      {r0-r12,lr}^
```

进入中断后，PC-4 的值自动存储在 R14_irq 中，则将其减 4 即可得到实际的返回地址，然后将寄存器 r0～r12 和寄存器 lr 入栈（注意，此时是入的 IRQ 模式的堆栈）。最后返回时将上述寄存器出栈即可，^表示将 SPSR_irq 复制到 SVC 模式的寄存器 CPSR 中。

读者可能有这样的疑问：发生中断时，并没有将 CPSR 寄存器的值存入到 IRQ 模式下的寄存器 SPSR_irq 中，为什么要恢复呢？请读者注意，发生中断时，CPSR 寄存器的值是由 ARM 处理器自动复制到 IRQ 模式下的 SPSR_irq 寄存器的（前文已经讲解过），但是中断返回时，需要用户手动恢复该寄存器的值。

技巧：这里之所以说有选择地保存上述寄存器的值的原因是，因为这几个寄存器是公用的，因此如果在中断处理程序中用到某个寄存器，则需要将其值保存起来，然后使用完后，在执行中断返回时，再将原来的值恢复。入栈操作需要耗费一定的时间，保存的寄存器越多，则中断响应时间就会延长。因此，如果在中断处理函数中仅仅是用到了寄存器 r0，则只需要将 r0 入栈即可，其他寄存器的值不需要入栈。如果在中断处理函数中用到了寄存器 r0～r8，则只需要将 r0～r8 入栈即可，其他寄存器的值不需要入栈。

提示：ARM 9 处理器的 5 级流水线结构是取指、译码、执行、访存、回写。取指部件负责从指令存储器读取指令；译码部件完成指令的译码；执行部件产生 ALU 运算结果或产生存储器地址；访存部件完成数据存储器的访存操作；回写部件负责将指令执行结果写回的寄存器。5 级流水线主要是把 3 级流水线中的执行单元进一步细化，减少了在每个时钟周期内必须完成的工作量，这样可以在一定程度上提高处理器的工作频率，此外，还具有独立的指令存储器和数据存储器，有效地避免了冲突的发生，CPI 明显减少。

下面总体回顾一下中断发生前后内存中的变化情况。

在系统启动后，PC 值自动设为 0x00000000，此处是复位异常中断的入口地址，根据第 7 章的讲解，接下来执行启动代码，然后跳转到用户主程序执行，如图 11-21 所示。注意，此时工作在 SVC 模式下，所以使用的是 SVC 模式下的堆栈，如图 11-21 中的灰色部分。

当中断发生后，又是什么样的情况呢？处理器执行完当前指令，会将 PC 值设为 0x00000018，如图 11-22 所示。注意，此时堆栈也发生了切换，如图 11-22 中的灰色部分所示。因此，在 IRQ 模式下使用的是 IRQ 模式下的堆栈。

注意：对于非计算机专业的读者，需要结合图 11-21 和图 11-22 理解 ARM 处理器工作模式切换的实质，重点理解工作模式切换过程中伴随的堆栈的切换。

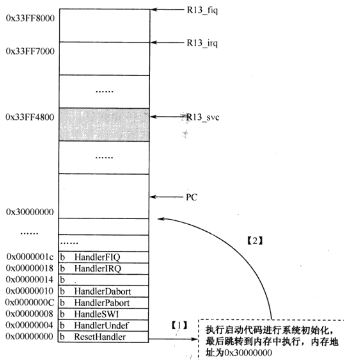


图 11-21　程序正常执行

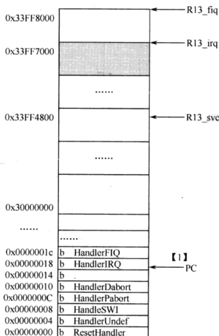


图 11-22　发生中断

11.2.4 中断处理流程引发的思考

经过前面一节的学习，读者对 ARM 中断处理流程有了基本的认识，但是还有以下问题没有解决。

- 用 C 语言写的中断处理程序是如何安装到向量表的呢？
- 每次中断发生时，都需要进行中断现场的保护和恢复。对这些问题，如果读者处理得不好，很可能就无法实现中断处理。既然用 C 语言可以加快开发速度，那么有没有好的办法使读者只需要注重具体怎么处理不同类型的中断，而完全不必考虑中断现场的保护和恢复呢？

下面结合启动代码部分，整体回顾中断处理的总体流程。

（1）当中断发生后，ARM 处理器执行完当前指令后，则将 PC 值设为 0x00000018，如图 11-23 所示。

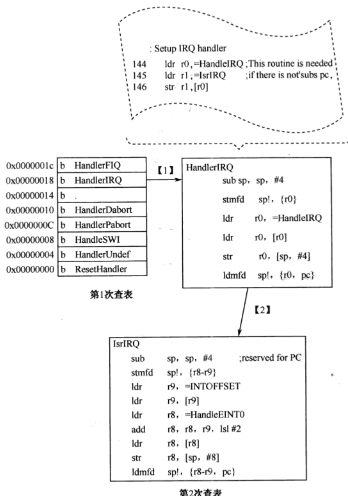


图 11-23 中断处理总体流程

（2）在 0x00000018 地址处存放的是一条跳转指令，则程序跳转到标号 HandlerIRQ 处执行，将 HandlerIRQ 宏调用展开，由于在系统启动阶段，HandleIRQ 地址处存放是 IsrIRQ 的地址（在汇编语言中，标号代表地址），则执行完 HandlerIRQ 后，PC 的值等于 IsrIRQ，程序跳转到 IsrIRQ 标号处执行。

（3）在 IsrIRQ 标号处，将 INTOFFSET 的值读入寄存器 r9，然后将寄存器 r8 指向第二级中断向量表的首地址，第二级中断向量表是在启动代码阶段用 MAP 和 FIELD 在内存中定义的。第二级中断向量表如表 11-3 所示。然后以 r9 的内容查表，将该地址处的值加载到 PC 就可以跳转到中断处理函数了。

小技巧：*因为每个函数的地址占 4 个字节，因此，需要将 r9 的值乘以 4。上述程序中是通过左移两位来实现将 r9 的值乘以 4。这里需要提醒初学者：尽量用左移运算来实现乘法可以提高运算的效率。同理，可以用右移运算来实现除法操作。*

表 11-3　第二级中断向量表

启动代码中的标号	内存地址	数　据
HandleReset	0×33FFFF00	
HandleUndef	0×33FFFF04	
HandleSWI	0×33FFFF08	
HandlePabort	0×33FFFF0C	
HandleDabort ← r8	0×33FFFF10	
HandleReserved	0×33FFFF14	
HandleIRQ	0×33FFFF18	
HandleFIQ	0×33FFFF1C	
HandleEINT0	0×33FFFF20	
HandleEINT1	0×33FFFF24	
HandleEINT2	0×33FFFF28	
HandleEINT3	0×33FFFF2C	
HandleEINT4_7	0×33FFFF30	
HandleEINT8_23	0×33FFFF34	
HandleCAM	0×33FFFF38	
HandleBATFLT	0×33FFFF3C	
HandleTICK	0×33FFFF34	
HandleWDT	0×33FFFF38	
……	……	
HandleADC	0×33FFFF80	

例 1：以外部中断 0 为例分析。

当外部中断 0 发生且没有被屏蔽时，寄存器 INTOFFSET 的值为 0。此时，

r8=0x33FFFF20，r9=0。因此，查表后得到的地址是 0x33FFFF20，将该地址处的值加载到 PC 即可。但是该地址处的值是什么呢？就是外部中断 0 中断处理函数的入口地址！当然这需要程序员将外部中断 0 中断处理函数的入口地址存放到该地址处。下面的代码在前文中已进行了讲解。

```
#define _ISR_STARTADDRESS        0x33ffff00
#define pISR_EINT0      (*(unsigned *)(_ISR_STARTADDRESS+0x20))
pISR_EINT0 = (unsigned int)Eint0_Isr ;

void __irq Eint0_Isr(void)
{
    Led1_On();Delay1s() ;Led1_Off() ;
    rSRCPND |= 1 << 0 ;//清除中断标志
    rINTPND |= 1 << 0 ;
}
```

例 2：以外部中断 2 为例分析。

当外部中断 2 发生且没有被屏蔽时，寄存器 INTOFFSET 的值为 2。此时，r8=0x33FFFF20，r9=2。因此，查表后得到的地址是 0x33FFFF28（0x33FFFF20+2*4），将该地址处的值加载到 PC 即可。当然，这需要提前将外部中断 2 中断处理函数的入口地址存放到该地址处。可用如下代码实现：

```
#define _ISR_STARTADDRESS        0x33ffff00
#define pISR_EINT2      (*(unsigned *)(_ISR_STARTADDRESS+0x28))
pISR_EINT2 = (unsigned int)Eint2_Isr ;
void __irq Eint2_Isr(void)
{
    Led2_On();Delay2s() ;Led2_Off() ;
    rSRCPND |= 1 << 0 ;//清除中断标志
    rINTPND |= 1 << 0 ;
}
```

最后一个问题是：前文讲到中断现场的保护和恢复，但是在上述中断处理函数中并没有发现入栈和出栈操作，也没有涉及中断返回地址的计算问题，这又是为什么呢？

细心的读者可能已经发现，在中断处理函数前面多了一个关键字：__irq（注意，前面是两个下画线）。对！就是这个关键字起到了巨大的作用，该关键字完成了上述所讲的中断现场的保护和恢复工作，可以从反汇编代码中看出一些端倪，使用__irq 关键字时外部中断 0 中断处理函数如下：

```
void __irq Eint0_Isr(void)
{
    Led1_On();Delay1s() ;Led1_Off() ;
    rSRCPND |= 1 << 0 ;//清除中断标志
    rINTPND |= 1 << 0 ;
}
```

反汇编之后的结果如图 11-24 所示。

```
Eint0_Isr
0x00000108:    e92d501f    .P-.    STMFD    r13!,{r0-r4,r12,r14}
0x0000010c:    e24dd004    .M.     SUB      r13,r13,#4
0x00000110:    e3a04456    VD..    MOV      r4,#0x56000000
0x00000114:    e5940014    ....    LDR      r0,[r4,#0x14]
0x00000118:    e3c00020     ...    BIC      r0,r0,#0x20
0x0000011c:    e5840014    ....    STR      r0,[r4,#0x14]
0x00000120:    ebfffffe    ....    BL       Delay1s
0x00000124:    e5940014    ....    LDR      r0,[r4,#0x14]
0x00000128:    e3800020     ...    ORR      r0,r0,#0x20
0x0000012c:    e5840014    ....    STR      r0,[r4,#0x14]
0x00000130:    e3a0044a    J...    MOV      r0,#0x4a000000
0x00000134:    e5901000    ....    LDR      r1,[r0,#0]
0x00000138:    e3811001    ....    ORR      r1,r1,#1
0x0000013c:    e5801000    ....    STR      r1,[r0,#0]
0x00000140:    e5901010    ....    LDR      r1,[r0,#0x10]
0x00000144:    e3811001    ....    ORR      r1,r1,#1
0x00000148:    e5801010    ....    STR      r1,[r0,#0x10]
0x0000014c:    e28dd004    ...     ADD      r13,r13,#4
0x00000150:    e8bd501f    .P..    LDMFD    r13!,{r0-r4,r12,r14}
0x00000154:    e25ef004    .^.     SUBS     pc,r14,#4
```

图 11-24 使用__irq 关键字时外部中断 0 中断处理函数反汇编

未使用__irq 关键字时外部中断 0 中断处理函数如下：

```
void  Eint0_Isr(void)
{
     Led1_On();Delay1s() ;Led1_Off() ;
     rSRCPND |= 1 << 0 ;//清除中断标志
     rINTPND |= 1 << 0 ;
}
```

反汇编之后的结果如图 11-25 所示。

```
Eint0_Isr
0x00000108:    e92d4010    .@-.    STMFD    r13!,{r4,r14}
0x0000010c:    e3a04456    VD..    MOV      r4,#0x56000000
0x00000110:    e5940014    ....    LDR      r0,[r4,#0x14]
0x00000114:    e3c00020     ...    BIC      r0,r0,#0x20
0x00000118:    e5840014    ....    STR      r0,[r4,#0x14]
0x0000011c:    ebfffffe    ....    BL       Delay1s
0x00000120:    e5940014    ....    LDR      r0,[r4,#0x14]
0x00000124:    e3800020     ...    ORR      r0,r0,#0x20
0x00000128:    e5840014    ....    STR      r0,[r4,#0x14]
0x0000012c:    e3a0044a    J...    MOV      r0,#0x4a000000
0x00000130:    e5901000    ....    LDR      r1,[r0,#0]
0x00000134:    e3811001    ....    ORR      r1,r1,#1
0x00000138:    e5801000    ....    STR      r1,[r0,#0]
0x0000013c:    e5901010    ....    LDR      r1,[r0,#0x10]
0x00000140:    e3811001    ....    ORR      r1,r1,#1
0x00000144:    e5801010    ....    STR      r1,[r0,#0x10]
0x00000148:    e8bd8010    ....    LDMFD    r13!,{r4,pc}
```

图 11-25 未使用__irq 关键字时外部中断 0 中断处理函数反汇编

对比图 11-24 和图 11-25 中的虚线框中的部分可以发现，使用__irq 关键字时，编译器会自动完成对寄存器的保存以及中断返回地址的计算，如图 11-24 中最后一行，但是不使

用__irq 关键字时，编译器并没有做上述事情。

经过上述讨论可以得出如下结论：

声明中断处理函数时，只需要使用__irq 关键字就可以实现中断现场的保存和恢复工作，用户可以将精力放在对具体中断的处理上，不需要关心中断现场的保存和恢复工作，这些都由编译器来完成。

__irq 关键字主要有以下作用：

- 中断发生后，自动保存所有需要保存的寄存器。
- 中断返回时，自动计算中断返回地址，并自动将 IRQ 模式下寄存器 SPSR_irq 的值恢复到寄存器 CPSR（中断进入什么模式，则将该模式下寄存器 SPSR 的值恢复到 CPSR 中）。

注意：很多编译器都支持__irq 关键字，读者可以仔细阅读相关编译器的使用说明。使用__irq 关键字在很大程度上加快了程序开发的效率。但是，使用__irq 关键字声明中断处理函数时，对中断处理函数有如下要求：函数不能有参数和返回值。在软中断实验中，将展示不使用__irq 关键字时如何编写中断处理函数。

11.2.5 实例测试

经过前面的讲解，已基本讲清与中断有关的问题，总体来说主要解决了以下几个问题。

- 发生中断后，CPU 何时开始响应中断：执行完当前指令，然后才会响应中断。
- 在响应中断过程中，ARM 处理器自动完成哪些工作，用户需要完成哪些工作。
- 中断现场如何保护和恢复。
- 如何写一个中断处理函数。
- __irq 关键字有什么作用。

下面结合上述代码，验证上述知识的正确性。

将上述代码编译，关于编译选项的设置，请读者参见第 6 章的相关内容，在此着重提出下面选项，如图 11-26 和图 11-27 所示。

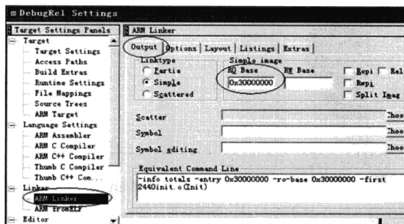


图 11-26 选择“ARM Linker”→“Output”

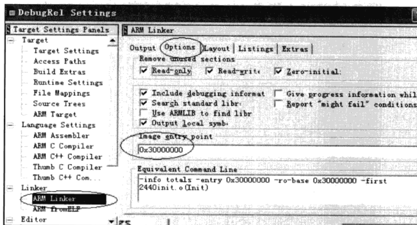


图 11-27　选择“ARM Linker”→“Options”

最后，编译后生成.bin 格式的二进制文件，然后从 NOR FLASH 启动，此时显示 U-Boot 启动界面，如图 11-28 所示（前提是，开发板的 NOR FLASH 中已经下载了 U-Boot，具体下载方法参见广州天嵌计算机科技有限公司开发板配套资料《TQ2440 开发板使用手册》）。

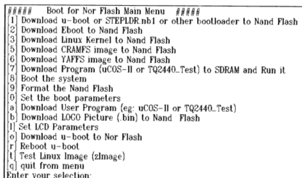


图 11-28　U-Boot 启动界面

接着，从键盘输入字母“a”。

然后，打开 DNW 软件，如图 11-29 所示，选择 USB Port\Transmit\Transimit。

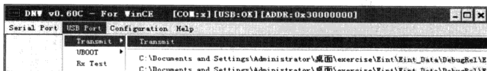


图 11-29　DNW 软件

此时弹出“打开”对话框，如图 11-30 所示，选择刚才生成的二进制文件即可将生成的二进制文件下载到 NAND FLASH 中（具体下载方法请读者参见广州天嵌计算机科技有限公司开发板配套资料《TQ2440 开发板使用手册》）。

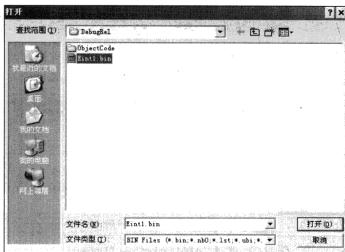


图 11-30 选择 Eint1.bin 文件

系统上电，选择从 NAND FLASH 启动，当按下 K1 时，LED1 亮；当按下 K2 时，LED2 亮；当按下 K3 时，LED3 亮；当按下 K4 时，LED4 亮。这说明，本实验已经达到了预期的效果。

11.2.6 为什么进入不了中断

在学习中断过程中，经常遇到的问题是进入不了中断，经常看到有很多朋友网上求助。下面将分析进入不了中断的原因以及解决方法。

在如图 11-28 所示的 U-Boot 启动界面中，从键盘输入数字“7”，则会将程序下载到内存地址 0x30000000 处，此时会出现如图 11-31 所示的界面。

然后，结合上面的讲解使用 DNW 软件，就可以将程序下载到 SDRAM 中，下载完成后，会自动启动，界面如图 11-32 所示，最后一行：Starting application at 0x30000000，即说明从 SDRANM 启动成功。

```
#####   Boot for Nor Flash Main Menu   #####
[1] Download u-boot or STEPLDR.nb1 or other bootloader to Nand Flash
[2] Download Eboot to Nand Flash
[3] Download Linux Kernel to Nand Flash
[5] Download CRAMFS image to Nand Flash
[6] Download YAFFS image to Nand Flash
[7] Download Program (uCOS-II or TQ2440_Test) to SDRAM and Run it
[8] Boot the system
[9] Format the Nand Flash
[0] Set the boot parameters
[a] Download User Program (eg: uCOS-II or TQ2440_Test)
[b] Download LOGO Picture (.bin) to Nand  Flash
[l] Set LCD Parameters
[o] Download u-boot to Nor Flash
[r] Reboot u-boot
[t] Test Linux Image (zImage)
[q] quit from menu
Enter your selection: 7
USB host is connected. Waiting a download.
```

图 11-31 从键盘输入数字“7”

```
#####    Boot for Nor Flash Main Menu    #####
[1] Download u-boot or STEPLDR.nb1 or other bootloader to  and Flash
[2] Download Eboot to Nand Flash
[3] Download Linux Kernel to Nand Flash
[5] Download CRAMFS image to Nand Flash
[6] Download YAFFS image to Nand Flash
[7] Download Program (uCOS-II or TQ2440_Test) to SDRAM an  Run it
[8] Boot the system
[9] Format the Nand Flash
[0] Set the boot parameters
[a] Download User Program (eg: uCOS-II or TQ2440_Test)
[b] Download LOGO Picture (.bin) to Nand  Flash
[l] Set LCD Parameters
[o] Download u-boot to Nor Flash
[r] Reboot u-boot
[t] Test Linux Image (zImage)
[q] quit from menu
Enter your selection: 7
USB host is connected. Waiting a download.

Now, Downloading [ADDRESS:30000000h,TOTAL:2074]
RECEIVED FILE SIZE:    2074 (2KB/S, 1S)
## Starting application at 0x30000000 ...
```

图 11-32　从 SDRAM 启动界面

此时在按下开发板上的 K1～K4 时，问题发生了！4 个 LED 没有任何反应。为什么呢？

现在，请读者明确这么一个问题：LED 不亮，说明并没有执行中断处理程序，但是中断发生没有呢？因为在前面实验中按下按键后确实能发生中断，然后在中断处理函数中将对应的 LED 点亮，所以按键没有问题（读者也可以用示波器来观察按键按下时电平的变化来确定一下）。既然能产生中断，中断处理程序又没有问题，则问题只可能出在一个地方：CPU 没找到中断处理函数！因此，中断虽然发生了，但是中断处理函数并没有执行。

要解决上面的问题，还得从 NOR FLASH 启动谈起。NOR FLASH 的容量是 2MByte，并且接在了 S3C2440 处理器的 BANK0，即从 0x00000000 开始的 2Mbyte 的地址范围内，如图 11-33 所示。

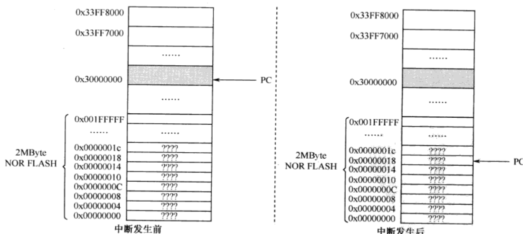


图 11-33　从 NOR FLASH 启动

请读者注意此时的情况，从 NOR FLASH 启动，但是 NOR FLASH 中已经烧写了 U-Boot（所谓的 U-Boot 就是通用的启动代码，读者可能注意到，本书第 7 章讲解了启动代码，但是这需要对处理器有一定的了解，也有一定的难度。因此，为了降低这一难度，有一部分人就写好了 U-Boot，读者只需要根据自己开发板的情况适当地裁剪就可以得到适合自己开发板的启动代码），经过 U-Boot 执行一系列的初始化工作后，最后跳转到 SDRAM 中执行

用户程序。

当中断发生时，ARM 处理器会将程序计数器 PC 的值设为 0x00000018，但是这个地址处存放的是什么呢？我们并不知道！（其实这里存放的是 U-Boot 对与 IRQ 中断的处理函数的入口地址）。因此，中断发生后，CPU 并不能找到上述中断处理函数，所以也就无法点亮对应的 LED。

怎么才能解决上述问题呢？还得从问题的原因入手！CPU 找不到中断处理函数，主要是因为在地址 0x00000018 处没有正确的跳转地址。因此，只需要将上述生成的二进制文件下载到 NOR FLASH 的 0x00000000 地址处即可。

注意：此时需要使用 H-JTAG 下载。

关于 H-JTAG 软件的设置请参见本书第 2 章，下面着重讲解程序的下载过程。

（1）打开 H-JTAG 软件，成功检测到 CPU 型号后，在“Flasher”菜单下选择“Start H-Flasher”，如图 11-34 所示。

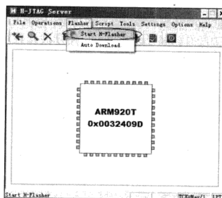


图 11-34　选择“Start H-Flasher”

（2）在弹出的 H-Flasher 窗口中，选择“Load”，然后选择“TQ2440_nor_eon.hfc”即可，如图 11-35 所示。

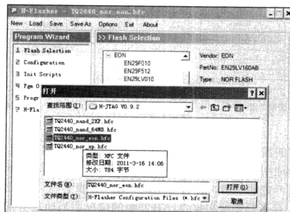


图 11-35　选择配置文件

说明：H-JATA 配置文件共有 4 个，具体选择哪一个配置文件，请参考具体的开发板硬件配置（如果读者对自己开发板的 FLASH 容量及型号不是很了解，请参见第 6 章扩展阅读部分）。

- TQ2440_nand_2KP.hfc：用于烧写 2KB 大页面 NAND FLASH 的 H-Flash 配置文件。
- TQ2440_nand_64MB.hfc：用于烧写 512B 页面 NAND FLASH 的 H-Flash 配置文件。
- TQ2440_nor_eon.hfc：用于烧写 NOR FLASH 的 H-Flash 配置文件。
- TQ2440_nor_sp.hfc：用于烧写 NOR FLASH 的 H-Flash 配置文件。

（3）在 H-Flasher 窗口中，选择“Flash Selection”，然后选择“EN29LV160AB”（不同的开发板可能具体型号不同，读者需要确定 Flash 芯片的型号），如图 11-36 所示。

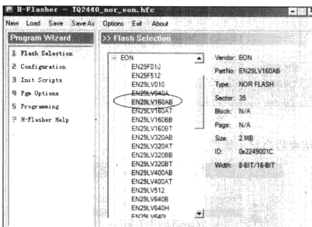


图 11-36　选择 FLASH 型号界面

（4）在 H-Flasher 窗口中，选择“Programming”，单击“Check”按钮，会显示出 Flash 型号，单击“Src File”右边的“…”按钮，会弹出打开对话框，然后找到“Eint1.bin”，最后单击“Program”按钮即可实现程序下载，整体过程如图 11-37、图 11-38 所示。

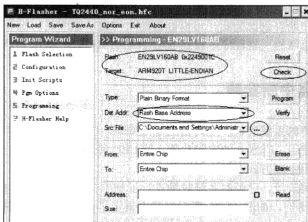


图 11-37　检测 Flash 型号

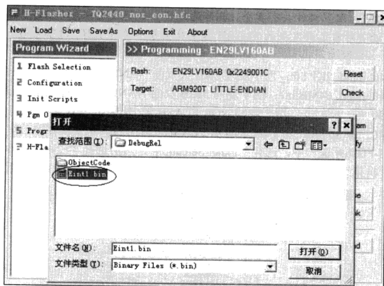


图 11-38 选择二进制文件

（5）程序下载结束后会显示如图 11-39 所示的界面。

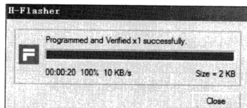


图 11-39 下载完成

系统上电，选择从 NOR FLASH 启动，当按下 K1 时，LED1 亮；当按下 K2 时，LED2 亮；当按下 K3 时，LED3 亮；当按下 K4 时，LED4 亮。这说明，上述问题的分析是正确的。

为什么进不了中断？到此为止，可以回答这个问题，原因是：没有正确安装中断向量表。解决方法可以总结为：将程序下载到 NAND FLASH 或者 NOR FLASH 中执行即可。

11.3 定时器中断实验

回顾第 8 章讲解的系统时钟和定时器的基础知识，在第 8 章中曾给出了如下实验。

实现的功能：使用定时器 0 的定时功能，使 LED 每秒钟闪烁一次，当时是使用查询方式实现的，现在使用中断方式实现上述功能。

因为在启动代码阶段，已经对系统时钟进行了初始化，PCLK=50 MHz，定时器的输入频率由 PCLK 分频得到，如图 11-40 所示向读者展示了从晶振输入频率到定时器工作频率产生的整体过程。

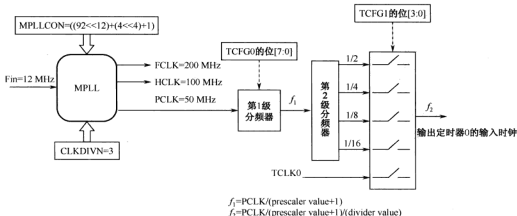
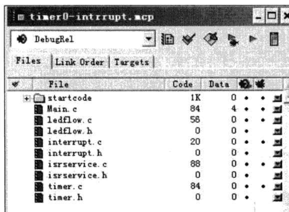


图 11-40 定时器 0 输入时钟产生过程

11.3.1 程序代码分析

定时器工程的文件布局如图 11-41 所示。



File	Code	Data
startcode	1K	0
Main.c	84	4
ledflow.c	56	0
ledflow.h	0	0
interrupt.c	20	0
interrupt.h	0	0
isrservice.c	88	0
isrservice.h	0	0
timer.c	84	0
timer.h	0	0

图 11-41 定时器工程的文件布局

该工程中的文件是在前面实验基础上修改得到的，ledflow.h 和 ledflow.c 文件内容没有变化，其他几个文件有变化，下面逐一展示。

定时器模块包含两个文件：timer.h 和 timer.c 文件。

timer.h 文件中声明了定时器 0 初始化函数 Timer0_Init()。

```
#ifndef _TIMER0_H_
#define _TIMER0_H_
void Timer0_Init(void) ;

#endif
```

timer.c 文件对定时器 0 初始化函数 Timer0_Init()进行了具体的实现。

```
    void Timer0_Init(void)
    {
1        rTCFG0     &=     ~(0xFF) ;
2        rTCFG0     |=     99 ;
3        rTCFG1     &=     ~(0xf) ;
4        rTCFG1     |=     0x02 ;
5        rTCNTB0    =      62500 ;   //1s 中断一次
6        rTCON      |=     (1 << 1) ;
7        rTCON      =      0x09 ;
    }
```

在此实验中，定时器 0 的输入时钟频率为 62.5 kHz，即定时器每秒钟计数 62 500 次。因此，初始化时定时器 0 初始值缓存寄存器中的值为 62 500，如第 5 行所示。

第 6 行，开启手动更新位，即当定时器开启后，TCNTB0 中的初始值会加载到内部寄存器 TCNT0 中。

第 7 行，关闭手动更新位，设置自动加载位，同时开启定时器，这样，TCNT0 进行减 1 计数，当 TCNT0 中的计数值减到 0 时，TCNTB0 中的数据会自动加载到 TCNT0 中并进行新一轮的减 1 计数。

中断初始化模块包含两个文件：interrupt.h 和 interrupt.c 文件。

```
interrupt.h 文件内容如下：
#ifndef __INTERRUPT_H__
#define __INTERRUPT_H__

void Timer0_Interrupt_Init(void) ;

#endif
interrupt.c 文件内容如下：
#include "2440addr.h"

void Timer0_Interrupt_Init(void)
{
    rINTMSK &= ~(1 << 10) ;
}
```

将定时器 0 中断屏蔽位设为无效。这样，当定时器 0 发生中断时，中断请求信号就可以顺利到达 CPU。

中断服务函数模块包含两个文件：isrservice.h 和 isrservice.c 文件。

isrservice.h 文件内容如下：

```
#ifndef __ISRSERVICE_h__
#define __ISRSERVICE_h__

void Isr_Init(void) ;
```

```
void __irq Timer0_Isr(void) ;

#endif
```

该文件主要声明了定时器 0 中断处理函数，在声明时使用了关键字__irq。这样，可以不必考虑中断现场的保护和恢复工作。

isrservice.c 文件内容如下：

```
#include"config.h"
#include"isrservice.h"

extern unsigned int flag ;

void Isr_Init(void)
{
	pISR_TIMER0 = (unsigned int)Timer0_Isr ;

}

void __irq Timer0_Isr(void)
{
	flag = !flag ;
	rSRCPND |= 1 << 10 ;
	rINTPND |= 1 << 10 ;
}
```

该文件主要是将定时器 0 中断处理函数加载到第 2 级中断向量表的对应地址（0x33FFFF48）处，第 2 级中断向量表如表 11-3 所示。

此外，还用 extern 关键字声明了一个外部变量 flag，该变量是在 Main.c 文件中定义的，当 1s 到来时，中断响应函数将该变量值取反，在主程序中通过检测该变量的值来实现不同的操作。

用户主函数 Main.c 文件中的内容如下：

```
#include "ledflow.h"
#include "isrservice.h"
#include "interrupt.h"
#include "timer.h"

void IO_Init() ;
unsigned int flag = 0 ;

int Main()
{
	IO_Init() ;
```

```
        while(1)
        {
                if(flag )
                {

                        Led2_On() ;
                }
                else
                {
                        Led2_Off() ;
                }
        }
        return 0;
}

void IO_Init()
{
        Led_Init() ;
        Timer0_Init() ;
        Timer0_Interrupt_Init() ;
        Isr_Init() ;
}
```

程序的基本原理如下。

程序开始进行了 LED 和定时器 0 的初始化，在定时器 0 初始化最后，打开了定时器 0，定时器 0 进行减 1 计数。

当 TCNT0 中的计数值减为 0 时，发生定时器 0 中断，然后调用定时器 0 中断响应函数，将变量 flag 的值取反。同时，TCNTB0 中的值会自动转入到 TCNT0 中，进行新一轮计数。在主循环中不断地检测变量 flag 的值，当该值为真时，点亮 LED2；当该值为假时，熄灭 LED2。

11.3.2 实例测试

编译生成.bin 格式的二进制文件，将其下载到 NAND FLASH 或者 NOR FALSH 中，启动开发板，此时 LED 已经闪烁起来了。

11.4 串口中断原理及实验

经过前面知识的讲解，相信读者对于 ARM 中断的处理流程有了大概的了解和认识。下面以 S3C2440 处理器为例，从宏观上讲解如何正确地使用中断。

11.4.1 如何正确使用中断

虽然这一节主要是讲解串口中断的使用，但是，笔者认为真正弄清楚中断处理的总体流程对于初学者尤其重要，实验必不可少，但也不是越多越好。下面讲解如何正确地使用中断，这部分知识主要是对上述知识的总结与概括，初学阶段不宜贪多，但是，要经一定量的实验来总结共性的东西，在已有知识的基础上找到问题的理论依据，这也是本书倡导的一个理念：提供的是机制而非策略，这样可以达到事半功倍的效果。

总体来说，正确使用中断需要以下 3 个步骤。

1．正确识别中断源，根据中断类型初始化中断

虽然中断源有很多，但是不外乎以下两种类型。

- 第 1 种：该类中断有很多子中断，如 UART0，有发送中断、接收中断和错误中断 3 个子中断。
- 第 2 种：该类中断没有子中断，如外部中断 0。

对于第 1 种类型的中断，初始化时需要将总中断屏蔽位置为无效，同时还需要将子中断屏蔽位也置为无效。

对于第 2 种类型的中断，由于没有子中断，因此，只需要将总中断屏蔽位置为无效即可。

例 1：初始化 UART0 的接收中断和发送中断，可以使用如下代码实现。

```
rINTMSK          &=  ~(1 << 28) ;
rINTSUBMSK       &=  ~((1 << 0) | (1 << 1)) ;
```

首先将 UART0 总中断屏蔽位置为无效，然后将发送中断和接收中断屏蔽位置为无效。

例 2：初始化定时器 0 中断，可以使用如下代码实现。

rINTMSK &= ～(1 << 10) ;

只需要将定时器 0 的中断屏蔽位置为无效即可。

2．编写中断处理函数

编写中断处理函数时需要使用关键字__irq。这样，在编译阶段，编译器会自动生成中断现场的保护和恢复时所需要的代码，用户只需要关注对于具体中断的处理即可。

在中断处理函数中需要将中断标志清除，方法是向相应的中断标志位写 1 即可。

- 对于第 1 种类型的中断，需要清除 SRCPND、INTPND 寄存器中相应的中断标志位，还需要清除 SUBSRCPND 中断相应的中断标志位。
- 对于第 2 种类型的中断，只需要清除 SRCPND、INTPND 寄存器中相应的中断标志位即可。

例 1：清除 UART0 发送中断标志和接收中断标志，可以使用如下代码实现。

```
  void __irq Uart0_Isr(void)
{
  if(rSUBSRCPND & (1 << 0))//进一步判断是接收中断
  {
        ……//在此处进行相应的中断处理

        rSUBSRCPND |= 1 << 0 ; //清除接收中断
```

```
    }
    if(rSUBSRCPND & (1 << 1))// 进一步判断是接收中断
    {
        ……//在此处进行相应的中断处理

        rSUBSRCPND |= 1 << 1 ;//清除发送中断标志
    }
    rSRCPND |= 1 << 28 ;//清除 UART0 总中断
    rINTPND |= 1 << 28 ;
}
```

例 2：清除定时器 0 中断标志。

```
    void __irq Timer0_Isr(void)
{
    ……//在此处进行中断处理

    rSRCPND |= 1 << 10 ;
    rINTPND |= 1 << 10 ;
}
```

3. 安装中断处理函数

因为在启动代码阶段已经在内存中定义好了中断向量表，所以安装中断处理函数只需要将中断处理函数的地址写入到中断向量表的对应地址即可。

在启动代码里，使用 MAP 和 FIELD 定义了第 2 级中断向量表，如图 11-42 所示。

标号	地址	数据
HandleEINT0	0x33FFFF20	
HandleEINT1	0x33FFFF24	
HandleEINT2	0x33FFFF28	
HandleEINT3	0x33FFFF2C	
HandleEINT4_7	0x33FFFF30	
HandleEINT8_23	0x33FFFF34	
HandleCAM	0x33FFFF38	
HandleBATFLT	0x33FFFF3C	
HandleTICK	0x33FFFF40	
HandleWDT	0x33FFFF44	
HandleTIMER0	0x33FFFF48	
HandleTIMER1	0x33FFFF4C	
HandleTIMER2	0x33FFFF50	
HandleTIMER3	0x33FFFF54	
HandleTIMER4	0x33FFFF58	
HandleUART2	0x33FFFF5C	
HandleLCD	0x33FFFF60	
HandleDMA0	0x33FFFF64	
HandleDMA1	0x33FFFF68	
HandleDMA2	0x33FFFF6C	
HandleDMA3	0x33FFFF70	
HandleMMC	0x33FFFF74	
HandleSPI0	0x33FFFF78	
HandleUART1	0x33FFFF7C	
HandleNFCON	0x33FFFF80	
HandleUSBD	0x33FFFF84	
HandleUSBH	0x33FFFF88	
HandleIIC	0x33FFFF8C	
HandleUART0	0x33FFFF90	
HandleSPI1	0x33FFFF94	
HandleRTC	0x33FFFF98	
HandleADC	0x33FFFF9C	

图 11-42　启动代码中使用汇编语言定义的中断向量表

注意：在汇编语言中，标号代表的是地址。因此，图 11-42 中最左边一列的标号和第 2 列的地址是等价的关系。现在的问题是：如何将中断函数的地址写入到中断向量表中对应的地址处呢？

在 2440addr.h 文件中，使用 define 定义的指针，指向这些地址处，如图 11-43 所示。

注意：这里的_ISR_STARTADDRESS=0x33FFFF00。因此，很容易计算出图 11-43 定义的指针和图 11-42 中的表项是一一对应的关系。

```
#define pISR_EINT0      (*(unsigned *)(_ISR_STARTADDRESS+0x20))
#define pISR_EINT1      (*(unsigned *)(_ISR_STARTADDRESS+0x24))
#define pISR_EINT2      (*(unsigned *)(_ISR_STARTADDRESS+0x28))
#define pISR_EINT3      (*(unsigned *)(_ISR_STARTADDRESS+0x2c))
#define pISR_EINT4_7    (*(unsigned *)(_ISR_STARTADDRESS+0x30))
#define pISR_EINT8_23   (*(unsigned *)(_ISR_STARTADDRESS+0x34))
#define pISR_CAM        (*(unsigned *)(_ISR_STARTADDRESS+0x38))
#define pISR_BAT_FLT    (*(unsigned *)(_ISR_STARTADDRESS+0x3c))
#define pISR_TICK       (*(unsigned *)(_ISR_STARTADDRESS+0x40))
#define pISR_WDT_AC97   (*(unsigned *)(_ISR_STARTADDRESS+0x44))
#define pISR_TIMER0     (*(unsigned *)(_ISR_STARTADDRESS+0x48))
#define pISR_TIMER1     (*(unsigned *)(_ISR_STARTADDRESS+0x4c))
#define pISR_TIMER2     (*(unsigned *)(_ISR_STARTADDRESS+0x50))
#define pISR_TIMER3     (*(unsigned *)(_ISR_STARTADDRESS+0x54))
#define pISR_TIMER4     (*(unsigned *)(_ISR_STARTADDRESS+0x58))
#define pISR_UART2      (*(unsigned *)(_ISR_STARTADDRESS+0x5c))
#define pISR_LCD        (*(unsigned *)(_ISR_STARTADDRESS+0x60))
#define pISR_DMA0       (*(unsigned *)(_ISR_STARTADDRESS+0x64))
#define pISR_DMA1       (*(unsigned *)(_ISR_STARTADDRESS+0x68))
#define pISR_DMA2       (*(unsigned *)(_ISR_STARTADDRESS+0x6c))
#define pISR_DMA3       (*(unsigned *)(_ISR_STARTADDRESS+0x70))
#define pISR_SDI        (*(unsigned *)(_ISR_STARTADDRESS+0x74))
#define pISR_SPI0       (*(unsigned *)(_ISR_STARTADDRESS+0x78))
#define pISR_UART1      (*(unsigned *)(_ISR_STARTADDRESS+0x7c))
#define pISR_NFCON      (*(unsigned *)(_ISR_STARTADDRESS+0x80))
#define pISR_USBD       (*(unsigned *)(_ISR_STARTADDRESS+0x84))
#define pISR_USBH       (*(unsigned *)(_ISR_STARTADDRESS+0x88))
#define pISR_IIC        (*(unsigned *)(_ISR_STARTADDRESS+0x8c))
#define pISR_UART0      (*(unsigned *)(_ISR_STARTADDRESS+0x90))
#define pISR_SPI1       (*(unsigned *)(_ISR_STARTADDRESS+0x94))
#define pISR_RTC        (*(unsigned *)(_ISR_STARTADDRESS+0x98))
#define pISR_ADC        (*(unsigned *)(_ISR_STARTADDRESS+0x9c))
```

图 11-43 在 2440addr.h 文件中定义指向该中断向量表中每一项的指针

例：#define pISR_Timer0 (*(unsigned *)_ISR_STARTADDRESS+0x48)。

这个定义相当于#define pISR_Timer0 (*(unsigned *)0x33FFFF48)，0x33FFFF48 仅仅是一个十六进制表示的数据而已，但是前面用（unsigned *）修饰，表示将 0x33FFFF48 强制转换为一个地址指针，即（unsigned *）0x33FFFF48 指向内存中地址 0x33FFFF48 处，准确地说是指向内存中从 0x33FFFF48 开始的连续的 4 个字节的内存片中（0x33FFFF48～0x33FFFF4B）。因此，（unsigned *）0x33FFFF48 其实就是（unsigned int *）0x33FFFF48 的缩写，如图 11-44 所示。

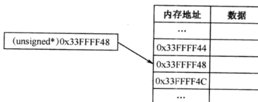


图 11-44 (unsigned *)0x33FFFF48 实例

然后在（unsigned *）0x33FFFF48 前面再加个*，这里的*是指针运算符（也叫“间接访问”运算符），表示取该内存单元中的数据。

最后再看#define pISR_Timer0 (*(unsigned *)_ISR_STARTADDRESS+0x48)，表示用#define 定义了一个新的类型 pISR_Timer0，pISR_Timer0 含义和(*(unsigned *)0x33FFFF48)完全相同，都表示访问内存单元 0x33FFFF48 中的数据。

因此，在程序中可以看到下面的语句：pISR_TIMER0 = (unsigned int)Timer0_Isr，上述语句相当于(*(unsigned *)0x33FFFF48) = (unsigned int)Timer0_Isr。

Timer0_Isr 是定时器 0 中断函数的入口地址（在 C 语言中，函数名也是代表了该函数的地址，又因为每个函数的入口地址占 4 个字节，因此使用了强制类型转换），其实就是向内存单元 0x33FFFF48 中写入了定时器 0 中断函数的入口地址，如图 11-45 所示。

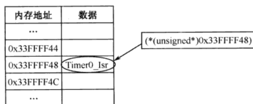


图 11-45 (*(unsigned *)0x33FFFF48)示意图

到此，中断函数的入口地址已经顺利地添加到中断向量表的对应地址处。

11.4.2 程序代码分析

有了上面的分析，下面应用上面的步骤，进行 UART0 中断实验。

本实验实现的基本功能：PC 通过 UART 发送一个字符到 S3C2440 处理器，S3C2440 处理器收到该字符后，将收到的字符再发送给 PC，用户通过超级终端或者串口调试助手观测数据是否正确即可。

本实验主要是在第 10 章“UART 基础实验”的基础上稍微做修改即可。

实验步骤如下：

（1）初始化串口，取消串口中断屏蔽位。

（2）编写串口中断处理函数。

（3）安装串口中断处理函数。

UART 中断实验工程的文件布局如图 11-46 所示。

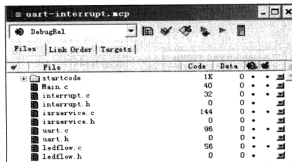


图 11-46 UART 中断实验工程的文件布局

UART 模块包含两个文件：uart.h 和 uart.c 文件。

uart.h 文件中声明了 UART 初始化函数 Uart0_Init()。

```
#ifndef __UART_H__
#define __UART_H__
extern void Uart0_Init(unsigned int baudrate) ;
#endif
uart.c 文件对上述函数进行了具体的实现。
#define PCLK 50000000                    //时钟源设为 PCLK
void Uart0_Init(unsigned int baudrate)
{
1    rGPHCON &= ~((3 << 4) | (3 << 6)) ;
2    rGPHCON |= ((2 << 4) | (2 << 6)) ;//GPH2--TXD[0];GPH3--RXD[0]
3    rGPHUP   = 0x00 ;
4    rULCON0 |= 0x03 ;                   //8 个数据位，1 个停止位
5    rUCON0   = 0x05 ;
6    rUBRDIV0 = (int) (PCLK / baudrate / 16)- 1;
7    rURXH0     = 0;
}
```

第 1～3 行，将 GPH2、GPH3 配置为 TXD、RXD 模式。

第 4 行，设置寄存器 ULCON0，设置数据发送格式为：8 个数据位，1 个停止位，无校验位。

第 5 行，发送模式和接收模式都使用查询方式。

第 6 行，设置波特率，其中波特率作为一个参数传递到该初始化函数。

第 7 行，将 URXH0 清零。

中断初始化模块包含两个文件：interrupt.h 和 interrupt.c 文件。

interrupt.h 文件内容如下：

```
#ifndef __INTERRUPT_H__
#define __INTERRUPT_H__


void Uart0_Interrupt_Init(void) ;

#endif
```

interrupt.c 文件内容如下：

```
#include "2440addr.h"

#include "2440addr.h"

void Uart0_Interrupt_Init(void)
```

```
{
	rINTMSK		&= ~(1 << 28) ;
	rINTSUBMSK	&= ~((1 << 0) | (1 << 1)) ;

}
```

UART0 中断属于第 2 类，因此需要将 UART0 总中断屏蔽位置为无效，然后将发送中断和接收中断屏蔽位置为无效。这样，在程序中才可以顺利地响应发送中断和接收。

中断服务函数模块包含两个文件：isrservice.h 和 isrservice.c 文件。

isrservice.h 文件内容如下：

```
#ifndef __ISRSERVICE_h__
#define __ISRSERVICE_h__

void Isr_Init(void) ;
void __irq Uart0_Isr(void) ;

#endif
```

该文件主要声明了 UART0 中断处理函数，在声明时使用了关键字__irq，这样可以不必考虑中断现场的保存和恢复工作。

isrservice.c 文件内容如下：

```
#include"config.h"
#include"isrservice.h"
#include"ledflow.h"
void Isr_Init(void)
{
1	pISR_UART0 = (unsigned int)Uart0_Isr ;
}

void __irq Uart0_Isr(void)
{
	unsigned char buf ;
	if(rSUBSRCPND & (1 << 0))//接收中断
	{
2		buf = rURXH0 ;
3		rUTXH0 = buf ;
4		Led1_On() ;
5		rSUBSRCPND |= 1 << 0 ;//清除接收中断
	}
	if(rSUBSRCPND & (1 << 1))//发送中断
	{
6		Led2_On() ;
```

```
7           rSUBSRCPND |= 1 << 1 ;
        }
8           rSRCPND |= 1 << 28 ;
9           rINTPND |= 1 << 28 ;
    }
```

第 1 行，主要是将 UART0 中断处理函数加载到第 2 级中断向量表的对应地址（0x33FFFF90）处。

第 2～3 行，当 UART0 收到一个字符后，会发生中断，此时，调用中断处理函数，在中断处理函数中进一步判断是发送中断还是接收中断。如果是接收中断，则将串口缓冲区的数据取出，然后将该数据写入发送缓冲区即可实现数据的发送。最后需要清除接收中断标志位。

为了便于读者形象地理解确实进入了 UART0 中断处理函数，在发送完接收中断处理函数中调用 Led1_On()函数点亮了 LED1。

第 6～7 行，当数据发送完后，产生发送中断，此时，还会进入该中断处理函数，由于此时是发送中断，因此第 2～5 行不执行，执行第 6～7 行，在此处也是点亮了一个 LED，然后清除发送中断标志位。

第 8～9 行，不管是发送中断还是接收中断，都需要清除 UART0 总中断标志。

Main.c 文件（用户主文件）内容如下：

```
#include "isrservice.h"
#include "interrupt.h"
#include "uart.h"
#include "ledflow.h"
#include "common.h"
void IO_Init() ;

int Main()
{
    IO_Init() ;

    while(1)
    {
        ;
    }
    return 0;
}

void IO_Init()
{
```

```
    Uart0_Init(115200);
    Uart0_Interrupt_Init();
    Isr_Init();
    Led_Init();
}
```

该文件主要完成基本的串口初始化、串口中断初始化和LED的初始化。在主循环中，什么也没有做，只是不断地循环，当发生接收中断时，自动进入中断处理，执行完中断处理后，再回到主循环不停地循环。

11.4.3 实例测试

编译、链接生成.bin 格式的二进制文件后下载到开发板，打开超级终端，波特率设为115 200，如图11-47所示，这是说明S3C2440成功接收到用户输入的数据，然后又将其发送到PC。

同时，也可观察到：当第1次按下键盘上的一个键时，开发板上的LED1和LED2亮了，这说明S3C2440收到了从PC发送的字符，并且进入了中断处理程序。

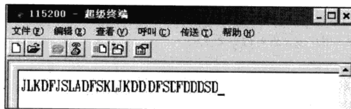


图11-47 串口中断实例测试

11.5 ARM中断之高级应用：软中断原理及实验

ARM处理器有7种工作模式。软中断，通俗地说就是为了从其他工作模式切换到管理模式（在管理模式，可以使用的资源最多），处理器提供软中断，主要是为了支持操作系统的系统功能调用，在一般应用中很少使用到软中断，当然在移植μC/OS-II操作系统时可以使用软中断来实现任务的切换。本节主要是为了向读者展示软中断的基本格式，以及如何实现软中断。

需要说明的是，本书启动代码主要工作在管理模式，所以发生软中断时，并没有发生处理器工作模式的切换。

11.5.1 程序代码分析

修改启动代码关于软中断的入口地址。

```
1    IMPORT C_Swi_Handler
```

```
2     ……
3     HandlerFIQ          HANDLER HandleFIQ
4     HandlerIRQ          HANDLER HandleIRQ
5     HandlerUndef        HANDLER HandleUndef
6     ;HandlerSWI         HANDLER HandleSWI
7     HandlerDabort       HANDLER HandleDabort
8     HandlerPabort       HANDLER HandlePabort

9     HandlerSWI                ;本实验中，进入软中断时，处理器本身就工作在管理模式下
10        stmfd       sp!,  {r0-r3,r12,lr}
11        ldr         r0,   [lr,#-4]
12        bic         r0,   r0,   #0xff000000
13        bl          C_Swi_Handler
14        ldmfd       sp!,  {r0-r3,r12,pc}^
```

在第 7 章讲解的启动代码的基础上，添加第 1 行，其中 C_Swi_Handler 是软中断处理函数的 C 语言入口地址。

注释掉第 6 行，在汇编语言中分号后面的内容都是注释。

添加第 9～14 行。由于软中断都有一个中断号，因此软中断处理函数需要一个参数，此时不能使用__irq 关键字声明软中断处理函数。

第 10、14 行主要是完成中断现场的保存和恢复工作。

第 11～12 行是计算中断向量号，将中断向量号放在寄存器 r0 中。

第 13 行调用 C 语言中实现的软中断处理函数，当然，此时寄存器 r0 中的值存储的是软中断号，该值作为 C_Swi_Handler 的参数。回顾第 5 章讲解的 ARM 处理器汇编语言和 C 语言混合编程，APCS 规则（ARM Process Call Standard）决定了在汇编语言中调用 C 语言中函数时，默认情况下寄存器 r0 存放第 1 个参数。

第 14 行是什么意思呢？请读者注意下面的分析。

当执行到软中断指令时，如图 11-48 所示，处理器并不是执行完当前指令再响应软中断，而是马上将处理器工作模式切换到管理模式。因此，此时 PC 值并没有更新（对比图 11-18 和图 11-19 中 PC 值更新的情况），发生软中断时，ARM 处理器自动将 PC-4 的值存入 R14_svc，而此值恰好就是中断返回时的地址。

因此得出的结论是：对于软中断，中断返回地址不需要调整。

最后一个问题是如何获取软中断号。在 ARM 处理器中，一般将软中断号放在发生软中断的那条指令的低 24 位。如何找到软中断号呢？

既然软中断号放在发生软中断的指令的低 24 位，则需要找到发生软中断的那条指令，因为发生软中断时，处理器会自动把 PC-4 的值存入寄存器 R14_svc，从图 11-48 中可以看到，发生软中断的那条指令就在 PC-8 处，所以只需要将 R14_svc 的值减去 4 就可以找到发生软中断的那条指令，则将其高 8 位屏蔽掉就可以找到对应的软中断号。

第 10～12 行就是用汇编指令实现的计算软中断号的代码。

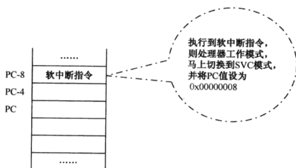


图 11-48　执行到软中断指令

下面讲解具体的软中断实验。

实验功能：在执行程序过程中，产生软中断，在软中断处理函数中将 LED 点亮。

软中断实验的文件布局如图 11-49 所示。

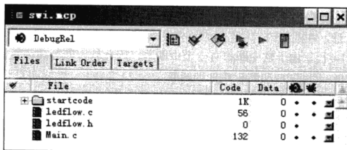


图 11-49　软中断实验的文件布局

ledflow.h 文件的内容如下：

```
#ifndef __LEDFLOW_H__
#define __LEDFLOW_H__
#include "2440addr.h"
#define Led1_On()      {rGPBDAT &= (~(1 << 5));}
#define Led1_Off()     {rGPBDAT |= (1 << 5);}
#define Led2_On()      {rGPBDAT &= (~(1 << 6));}
#define Led2_Off()     {rGPBDAT |= (1 << 6);}
#define Led3_On()      {rGPBDAT &= (~(1 << 7));}
#define Led3_Off()     {rGPBDAT |= (1 << 7);}
#define Led4_On()      {rGPBDAT &= (~(1 << 8));}
#define Led4_Off()     {rGPBDAT |= (1 << 8);}
extern void Led_Init(void);

#endif
```

ledflow.c 文件的内容如下：

```
#include "ledflow.h"
#include "2440addr.h"

void Led_Init(void)
{
      rGPBCON &= ~((3 << 10) | (3 << 12) | (3 << 14) |(3 << 16));
      rGPBCON |= ((1<<10) | (1<<12) | (1<<14) | (1<<16)) ;
      rGPBUP   &= ~((1 << 5) | (1 << 6) | (1 << 7) || (1 << 8) ) ;
      rGPBDAT |= (1 << 5) | (1 << 6) | (1 << 7) | (1 << 8) ;
}
```

Main.c 文件的内容如下：

```
#include "ledflow.h"

void IO_Init() ;
1     extern void C_Swi_Handler(unsigned num) ;

2     __swi(0x01) void led1(void);
3     __swi(0x02) void led2(void);
4     __swi(0x03) void led3(void);
5     __swi(0x04) void led4(void);

int Main()
{
      IO_Init() ;
      while(1)
      {
6           led1() ;
7           led2() ;
8           led3() ;
9           led4() ;
      }
      return 0;
}

void IO_Init()
{
      Led_Init() ;
}
```

```
void   C_Swi_Handler(unsigned num)
{
    switch(num)
    {
        case   0x01:
               Led1_On() ;   break ;
        case   0x02:
               Led2_On() ;   break ;
        case   0x03:
               Led3_On() ;   break ;
        case   0x04:
               Led4_On() ;   break ;
        default          :
               break ;
    }
}
```

第 1 行，用 extern 关键字声明了一个外部函数，因为此函数在 2440init.s 中要用到。

第 2～5 行，用__swi 关键字声明了 4 个软中断函数。__swi 关键字括号中的数字表明了该函数对应的软中断号。例如，调用 led1()，将产生软中断，该软中断的中断号是 0x01。

第 6～9 行，调用上述函数，则会产生相应的软中断。在前文讲到，软中断号存储在发生软中断的指令的低 24 位，这时可以通过反汇编看一下，如图 11-50 所示，虚线框中对应的是该指令的指令码。可见，低 24 位恰好对应软中断号。

```
Main
    0x00000004:      e92d4008     .@-.     STMFD     r13!,{r3,r14}
    0x00000008:      ebfffffe     ....     BL        IO_Init    ; 0x0
    0x0000000c:      ef000001     ....     SWI       0x1
    0x00000010:      ef000002     ....     SWI       0x2
    0x00000014:      ef000003     ....     SWI       0x3
    0x00000018:      ef000004     ....     SWI       0x4
    0x0000001c:      eafffffa     ....     B         {pc} - 0x10   ; 0xc
```

图 11-50　软中断指令反汇编

软中断执行过程如图 11-51 所示。

第一步：程序执行过程中遇到函数 led1()，则产生软中断，原因是前面已经用__swi(0x01) void led1(void)进行了声明。

第二步：处理器进行工作模式切换，并将 PC 值设为 0x00000008，执行启动代码中的软中断处理，找出软中断号，作为参数，然后调用 C 语言实现的软中断处理函数 C_Swi_Handler。

第三步：执行软中断处理函数，点亮相应的 LED。

第四步：从 C_Swi_Handler 返回到调用处。

第五步，执行软中断返回，返回到产生软中断的地方，接着执行程序。

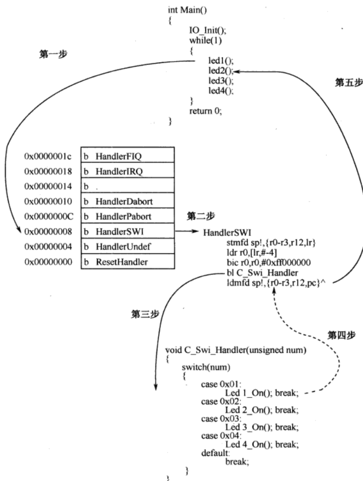


图 11-51 软中断执行过程

11.5.2 实例测试

编译、链接生成.bin 格式的二进制文件后下载到开发板，可看到，4 个 LED 已经被点亮，这说明已经正确执行了软中断。

上述实验只是讲解了软中断的基本实现，软中断的概念需要了解一下即可，在操作系统移植阶段可以进行深入的学习。

注意：经过本章对中断的讲述，虽然触及了 ARM 中断处理的基础知识，但是如果读者想深入学习中断，达到深层次应用的目的，则需要真正掌握位置无关代码（PIC）和可重入函数。关于位置无关代码和可重入函数，在此不做过多的讲述，有兴趣的读者可以查找相关书籍进行学习。

11.5.3 软中断所用到的启动代码

初学者在学习过程中，往往很小的一点改动就会被困扰好长时间。因此，为了读者查阅方便，将该实验所用到的完整的启动代码展示在这里。读者可以将其与第 7 章的启动代码对比一下，在前文中已经将改动的地方讲解过，改动的地方用加粗字体表示出来了。

```
        GET option.inc
        GET memcfg.inc
        GET 2440addr.inc
; constants defination
    USERMODE            EQU     0x10
    FIQMODE             EQU     0x11
    IRQMODE             EQU     0x12
    SVCMODE             EQU     0x13
    ABORTMODE           EQU     0x17
    UNDEFMODE           EQU     0x1b
    MODEMASK            EQU     0x1f
    NOINT               EQU     0xc0

    ;The location of stacks
    UserStack    EQU    (_STACK_BASEADDRESS-0x3800)    ;0x33ff4800 ~
    SVCStack     EQU    (_STACK_BASEADDRESS-0x2800)    ;0x33ff5800 ~
    UndefStack   EQU    (_STACK_BASEADDRESS-0x2400)    ;0x33ff5c00 ~
    AbortStack   EQU    (_STACK_BASEADDRESS-0x2000)    ;0x33ff6000 ~
    IRQStack     EQU    (_STACK_BASEADDRESS-0x1000)    ;0x33ff7000 ~
    FIQStack     EQU    (_STACK_BASEADDRESS-0x0)       ;0x33ff8000 ~

        MACRO
    $HandlerLabel HANDLER $HandleAddr
    $HandlerLabel
        sub     sp, sp, #4
        stmfd   sp!, {r0}
        ldr     r0, =$HandleAddr
        ldr     r0, [r0]
        str     r0, [sp, #4]
        ldmfd   sp!, {r0, pc}
        MEND

        IMPORT  |Image$$RO$$Base|       ; Base of ROM code
        IMPORT  |Image$$RO$$Limit|      ; End of ROM code (=start of ROM data)
        IMPORT  |Image$$RW$$Base|       ; Base of RAM to initialise
        IMPORT  |Image$$ZI$$Base|       ; Base and limit of area
        IMPORT  |Image$$ZI$$Limit|      ; to zero initialise
```

```
    IMPORT   Main                          ; The main entry of mon program
    IMPORT   RdNF2SDRAM                    ; Copy Image from Nand Flash to SDRAM
    IMPORT C_Swi_Handler                   ; 添加了这一行

    AREA     Init, CODE, READONLY
    ENTRY
ResetEntry
    b      ResetHandler
    b      HandlerUndef     ;handler for Undefined mode
    b      HandlerSWI       ;handler for SWI interrupt
    b      HandlerPabort    ;handler for PAbort
    b      HandlerDabort    ;handler for DAbort
    b      .                ;reserved
    b      HandlerIRQ       ;handler for IRQ interrupt
    b      HandlerFIQ       ;handler for FIQ interrupt


HandlerFIQ           HANDLER HandleFIQ
HandlerIRQ           HANDLER HandleIRQ
HandlerUndef         HANDLER HandleUndef
; HandlerSWI         HANDLER HandleSWI     ; 注释掉了这一行
HandlerDabort        HANDLER HandleDabort
HandlerPabort        HANDLER HandlePabort

HandlerSWI           ; 添加了如下 6 行
        stmfd        sp!,   {r0-r3,r12,lr}
        ldr          r0,    [lr,#-4]
        bic          r0,    r0,   #0xff000000
        bl           C_Swi_Handler
        ldmfd        sp!,   {r0-r3,r12,pc}^
IsrIRQ
    sub        sp, sp, #4            ;reserved for PC
    stmfd      sp!, {r8-r9}
    ldr        r9, =INTOFFSET
    ldr        r9, [r9]
    ldr        r8, =HandleEINT0
    add        r8, r8, r9, lsl #2
    ldr        r8, [r8]
    str        r8, [sp, #8]
    ldmfd      sp!, {r8-r9, pc}

    LTORG
```

```
ResetHandler
        ldr   r0, =WTCON              ;watch dog disable
        ldr   r1, =0x0
        str   r1, [r0]

        ldr   r0, =INTMSK
        ldr   r1, =0xffffffff         ;all interrupt disable
        str   r1, [r0]

        ldr   r0, =INTSUBMSK
        ldr   r1, =0x7fff             ;all sub interrupt disable
        str   r1, [r0]

        ldr   r0, =LOCKTIME
        ldr   r1, =0xffffff
        str   r1, [r0]

        ldr   r0, =CLKDIVN
        ldr   r1, =CLKDIV_VAL
        str   r1, [r0]

        [ CLKDIV_VAL>1              ;means Fclk:Hclk is not 1:1.
        mrc  p15, 0, r0, c1, c0, 0
        orr   r0, r0, #0xc0000000
        mcr  p15, 0, r0, c1, c0, 0
        |
        mrc p15, 0, r0, c1, c0, 0
        bic r0, r0, #0xc0000000
        mcr p15, 0, r0, c1, c0, 0
        ]
            ;Configure UPLL
        ldr   r0, =UPLLCON
;Fin = 12.0MHz,   UCLK = 48MHz
        ldr   r1, =((U_MDIV<<12)+(U_PDIV<<4)+U_SDIV)
        str   r1, [r0]
        nop        ; at least 7-clocks delay must be inserted for setting hardware be completed.
        nop
        nop
        nop
        nop
        nop
        nop
            ;Configure MPLL
```

```
        ldr  r0, =MPLLCON
             ;Fin = 12.0MHz,  FCLK = 200MHz
        ldr  r1, =((M_MDIV<<12)+(M_PDIV<<4)+M_SDIV)
        str  r1, [r0]
;Set memory control registers
        adrl        r0,  SMRDATA            ;please cautionl!
        ldmia       r0,  {r1-r13}
        ldr         r0,  =BWSCON
        stmia       r0,  {r1-r13}

        bl    InitStacks
;***********************************************************************

        ldr         r0,  =BWSCON
        ldr         r0,  [r0]
        ands        r0,  r0,  #6          ;OM[1:0] != 0,  NOR FLash boot
        bne         copy_proc_beg         ;do not read nand flash
        adr         r0,  ResetEntry       ;OM[1:0] == 0,  NAND FLash boot
        cmp         r0,  #0               ;if use Multi-ice,
        bne         copy_proc_beg         ;do not read nand flash for boot
;***********************************************************************
nand_boot_beg
        bl    RdNF2SDRAM
        ldr   pc,  =copy_proc_beg
;***********************************************************************
copy_proc_beg
        adr         r0,  ResetEntry
        ldr         r2,  BaseOfROM
        cmp         r0,  r2
        ldreq       r0,  TopOfROM
        beq         InitRam
        ldr r3,     TopOfROM
0
        Ldmia       r0!,  {r4-r7}
        stmia       r2!,  {r4-r7}
        cmp         2,  r3
        bcc         %B0

        sub         r2,  r2,  r3
        sub         r0,  r0,  r2

InitRam
        ldr         r2,  BaseOfBSS
```

```
        ldr     r3,  BaseOfZero
0
        cmp     r2,  r3
        ldrcc   r1,  [r0],  #4
        strcc   r1,  [r2],  #4
        bcc     %b0

        mov     r0,  #0
        ldr     r3,  EndOfBSS
1
        cmp     r2,  r3
        strcc   r0,  [r2],  #4
        bcc     %b1
; Setup IRQ handler
        ldr  r0, =HandleIRQ   ;This routine is needed
        ldr  r1, =IsrIRQ      ;if there is not 'subs pc, lr, #4' at 0x18,  0x1c
        str  r1, [r0]
        b    Main       ;Do not use main() because ......

InitStacks
        mrs  r0, cpsr
        bic  r0, r0, #MODEMASK
        orr  r1, r0, #UNDEFMODE|NOINT
        msr  cpsr_cxsf, r1    ;UndefMode
        ldr  sp, =UndefStack        ; UndefStack=0x33FF_5C00

        orr  r1, r0, #ABORTMODE|NOINT
        msr  cpsr_cxsf, r1    ;AbortMode
        ldr  sp, =AbortStack        ; AbortStack=0x33FF_6000

        orr  r1, r0, #IRQMODE|NOINT
        msr  cpsr_cxsf, r1    ;IRQMode
        ldr  sp, =IRQStack          ; IRQStack=0x33FF_7000

        orr  r1, r0, #FIQMODE|NOINT
        msr  cpsr_cxsf, r1          ;FIQMode
        ldr  sp, =FIQStack          ; FIQStack=0x33FF_8000

        bic  r0, r0, #MODEMASK|NOINT
        orr  r1, r0, #SVCMODE
        msr  cpsr_cxsf, r1          ;SVCMode
        ldr  sp, =SVCStack          ; SVCStack=0x33FF_5800
        mov  pc, lr
```

```
        ltorg

SMRDATA
        DCD 0x22011000
        DCD 0x00000700      ;GCS0
        DCD 0x00000700      ;GCS1
        DCD 0x00000700      ;GCS2
        DCD 0x00000700      ;GCS3
        DCD 0x00000700      ;GCS4
        DCD 0x00000700      ;GCS5
        DCD ((B6_MT<<15)+(B6_Trcd<<2)+(B6_SCAN))      ;GCS6
        DCD ((B7_MT<<15)+(B7_Trcd<<2)+(B7_SCAN))      ;GCS7
        DCD ((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Tsrc<<18)+REFCNT)
        DCD 0xB1
        DCD 0x30        ;MRSR6 CL=3clk
        DCD 0x30        ;MRSR7 CL=3clk

BaseOfROM   DCD     |Image$$RO$$Base|
TopOfROM    DCD     |Image$$RO$$Limit|
BaseOfBSS   DCD     |Image$$RW$$Base|
BaseOfZero  DCD     |Image$$ZI$$Base|
EndOfBSS    DCD     |Image$$ZI$$Limit|

        ALIGN

AREA RamData， DATA， READWRITE

        ^   _ISR_STARTADDRESS               ; _ISR_STARTADDRESS=0x33FF_FF00
HandleReset         #   4
HandleUndef         #   4
HandleSWI           #   4
HandlePabort        #   4
HandleDabort        #   4
HandleReserved      #   4
HandleIRQ           #   4
HandleFIQ           #   4


;IntVectorTable  ;@0x33FF_FF20
HandleEINT0         #   4
HandleEINT1         #   4
HandleEINT2         #   4
HandleEINT3         #   4
```

```
HandleEINT4_7       #    4
HandleEINT8_23      #    4
HandleCAM           #    4
HandleBATFLT        #    4
HandleTICK          #    4
HandleWDT           #    4
HandleTIMER0        #    4
HandleTIMER1        #    4
HandleTIMER2        #    4
HandleTIMER3        #    4
HandleTIMER4        #    4
HandleUART2         #    4
HandleLCD           #    4;@0x33FF_FF60
HandleDMA0          #    4
HandleDMA1          #    4
HandleDMA2          #    4
HandleDMA3          #    4
HandleMMC           #    4
HandleSPI0          #    4
HandleUART1         #    4
HandleNFCON         #    4
HandleUSBD          #    4
HandleUSBH          #    4
HandleIIC           #    4
HandleUART0         #    4
HandleSPI1          #    4
HandleRTC           #    4
HandleADC           #    4
    END
```

11.6 本章小结

本章主要讲解了 ARM 中断系统，首先讲解了中断的基本概念，然后重点分析了中断响应的过程。中断响应的过程涉及 ARM 处理器工作模电的切换、堆栈的切换以及中断返回地址的计算等知识，这是学习 ARM 处理器的重点也是难点，希望通过本章的学习，读者可以对 ARM 中断系统有个基本的认识与理解。

如果初学者感觉本章较难，推荐读者多读几遍，毕竟本章涉及了中断的大部分知识，有一定的难度。

第 12 章

NAND FLASH 原理与实验

在嵌入式系统设计中，NAND FLASH 是一种常见的数据存储设备，尤其是系统需要用到大量数据的存储时，经常选用 NAND FLASH。本章对 NAND FLASH 的基本组成、硬件接口、访问时序以及基本的页操作进行讲解。

12.1 FLASH 概述

常见的 FLASH 主要有 NOR FLASH 和 NAND FLASH 两种：NAND FLASH 不能执行代码，主要用于存储大量数据；NOR FLASH 中的数据掉电不丢失，而且可以执行程序，常用于存储系统的启动代码。

NAND 结构能提供极高的单元密度，可以达到高存储密度，并且写入和擦除的速度也很快。应用 NAND 的困难在于 FLASH 的管理需要特殊的系统接口。

Intel 公司于 1988 年首先开发出 NOR FLASH 技术。NOR 的特点是芯片内执行（XIP, eXecute In Place），应用程序可以直接在 NOR FLASH 内运行，不必再把代码读到系统 RAM 中。NOR FLASH 的传输效率很高，在 1～4 MB 的小容量时具有很高的成本效益，但是很低的写入和擦除速度大大影响了它的性能。

NAND FLASH 与 NOR FLASH 的性能对比如表 12-1 所示。

表 12-1 NAND FLASH 与 NOR FLASH 的性能对比

		NAND FLASH	NOR FLASH
容量		16～256MByte	1～32MByte
能否执行程序		否	是
可靠性		较低	较高
读/写性能	写	快	慢
	读	较快	较快
平均擦写次数		10～100 万次	1～10 万次
接口方式		I/O 接口，有时需要专门的控制器	与 RAM 接口一致
访问模式		顺序访问	随机访问
主要用途		存储数据	保存启动代码和系统固件
价格		较低	较高

为什么NAND FLASH无法执行程序呢？这主要是由于NAND FLASH的接口主要包括几个I/O口，对其中数据的访问是串行的访问，无法实现随机访问，因此NAND FLASH无法执行程序。

12.1.1 NAND FLASH的基本结构

笔者采用的TQ2440开发板上的NAND FLASH芯片型号是K9F2G08U0B，在谷歌网站输入K9F2G08U0B.pdf即可找到其对应的数据手册。

因为NAND FLASH接口电路是通过NAND FLASH控制器与S3C2440处理器相接的，因此，读者不需要关心具体的访问时序，之所以提供NAND FLASH控制器就是为了便于使用NAND FLASH。如果没有NAND FLASH控制器，则需要产生相应的访问时序。

下面先介绍NAND FLASH引脚以及内部存储器组织结构。

NAND FLASH（K9F2G08U0B）引脚如表12-2所示。

表12-2 NAND FLASH（K9F2G08U0B）引脚

引脚名称	功能描述
$I/O_0 \sim I/O_7$	数据/命令输入
CLE	命令锁存信号
ALE	地址锁存信号
$\overline{CE}$	片选信号
$\overline{RE}$	读使能信号
$\overline{WE}$	写使能信号
$\overline{WP}$	写保护信号
$R/\overline{B}$	就绪/忙指示信号
Vcc	电源
Vss	地

NAND FLASH（K9F2G08U0B）内部存储器组织结构如图12-1所示。

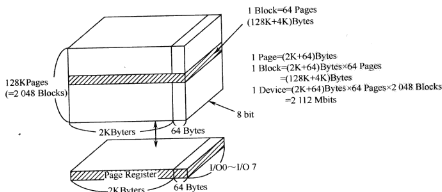


图12-1 NAND FLASH（K9F2G08U0B）内部存储器组织结构

NAND FLASH的存储器组成主要有两个部分：页（Page）、块（Block）。

- 每页大小为：2 K+64 字节， 2K 字节用来存储数据，64 字节主要用于存储控制信息，如图 12-1 所示。每一页的末尾都有 64 字节的额外空间，主要是为了便于管理每一页。

例如，使用这 64 位中的某一位表示该页是否已经写满数据的标志位。如果该位为 1，则表示该页写满。如果该位为 0，则表示该页为空。只需要查询该位即可知道该页是否存满数据。

- 每块大小为：64 个页。

整个 NAND FLASH 由 2 K（2048）个块组成。

因此，从数据输出可以看到 K9F2G08U0B 的容量是 256 MBytes×8Bit，这是怎么算出来的呢？

NAND FLASH 容量=块的数目×每块的容量

=块的数目×（每块包含页的数目×每页的容量）

=2 K 块×(64 页×(2 KBytes+64 Bytes))

=2 K 块×64 页×2 KBytes+2K 块×64 页×64 Bytes

=256 MBytes +8 MBytes

注意：1K（Bytes）=1 024 字节，1 M（Bytes）=1 K（Bytes）×1 K（Bytes）。

经过上面的计算，可以得到，前面的 256 M 表示该 NAND FLASH 可以存储 256 M 个字节数据，后面的 8 M 字节的数据主要用于保存每一页的控制信息。

小技巧：NAND FLASH 完全可以用一个小区来比喻。

整个小区由一栋栋楼房组成，每栋楼分几个楼层，每层楼由不同的房间组成。

NAND FLASH 类似于整个小区，每个块（Block）类似于小区里面的一栋楼，每个页（Page）类似于一层楼。

前文讲到，每页由 2 K+64 字节组成，这 2 K 字节用来存储数据的，最末尾的 64 个字节主要用于存储控制信息。类似的，每层楼有 2 K+64 个房间，其中，2 K 个房间是用来住的，剩下的 64 个房间是值班室。

12.1.2 NAND FLASH 接口电路

因为 S3C2440 处理器内部已经集成了 NAND FLASH 控制器。因此，接口电路将变得很简单，只需要将 S3C2440 处理器的对应引脚和 NAND FLASH 的对应引脚接上即可，如图 12-2 所示。

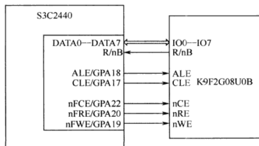


图 12-2 NAND FLASH 接口电路

现在的问题是：不同的 NAND FLASH 的容量不一样，接口线宽不一样，S3C2440 处理器如何获得这些信息呢？既然是 NAND FLASH 控制器，就必须处理好这些事情。在讲解之前，先对 NAND FLASH 控制器的作用进行简单的总结：

- 提供外接 NAND FLASH 的接口信息，包括接口线宽、容量等信息。
- 提供访问 NAND FLASH 所需的时序信息。

S3C2440 处理器提供了如下方法来识别 NAND FLASH 接口线宽和容量。

S3C2440 处理器提供了 OM[1:0]、NCON、GPG13、GPG14 和 GPG15 共 5 个信号来选择 NAND FLASH 启动。

- OM[1:0]：在启动代码中读取这两个引脚电平的值，当这两个引脚电平均为低电平时，从 NAND FLASH 启动（回顾第 7 章的启动代码中的第 106～112 行）。
- NCON：NAND FLASH 的类型选择信号。
  0：正常型 NAND FLASH（页面大小：256 Words/512 Bytes，3/4 地址周期）。
  1：高级型 NAND FLASH（页面大小：1 KWords/2 KBytes，4/5 地址周期）。
- GPG13：NAND FLASH 页容量选择信号。
  0：=256 Words（NCON=0）或=1 KWords（NCON= 1）。
  1：=512 Bytes（NCON= 0）或=2 KBytes（NCON=1）。
- GPG14：NAND FLASH 地址周期选择。
  0：3 地址周期（NCON= 0）或 4 地址周期（NCON= 1）。
  1：4 地址周期（NCON= 0）或 5 地址周期（NCON= 1）。
- GPG15：NAND FLASH 接口线宽选择。
  0：8 位总线宽度。
  1：16 位总线宽度。

下面看一下 TQ2440 开发板上的这部分接口电路，NAND FLASH 配置信号实例如图 12-3 所示。

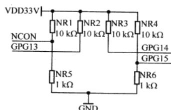


图 12-3　NAND FLASH 配置信号实例

从图 12-3 中可以看到，下面有个注释：“当 XXX 为 F1G/F2G/F4G 或 K8G 时，NR5 不焊”，这是什么意思呢？这里是工程师在画电路时考虑到将来可能会使用不同型号的 NAND FLASH，因此设计了上述电路，笔者结合自己使用的开发板讲解一下上述注释的意思。笔者使用的开发板 NAND FLASH 型号为 K9F2G08U0B，也就是这里的 F2G，则 NR5 不需要焊接。到底焊接还是没有焊接呢？下面看一下开发板上的实物图，如图 12-4 所示，可见，电阻 NR5 确实没有焊接。

电阻 NR5 不焊接，结合图 12-3 可以得出上述信号的电平关系：NCON=1、GPG13=1、GPG14=1、GPG15=0。

结合前面的讨论，可以得出如下结论：

- NCON=1，表示 NAND FLASH 为高级型 FLASH，页面大小是 1 KWords/2 KBytes。

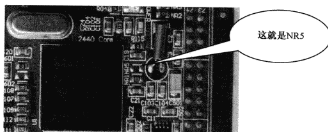


图 12-4　开发板上的 NR5

- GPG13=1，表示该信号和 NCON 信号共同决定了 NAND FLASH 页面的大小，即页面大小是 2 KBytes。
- GPG14=1，表示访问的地址周期为 5 个地址周期。
- GPG15=0，表示接口线宽是 8 位，如图 12-2 中 IO0～IO7 共 8 根线。

注意：上述推导过程仅仅是为了帮助初学者尽快熟悉 NAND FLASH 的配置方法。读者在实际设计硬件时需要根据 NAND FLASH 的情况对上述信号进行正确配置。

12.1.3　如何访问 NAND FLASH

S3C2440 处理器已经集成了 NAND FLASH 控制器，因此对 NAND FLASH 的访问将变得很简单，对 NAND FLASH 的访问操作（包括读、写和擦除 3 种操作）只需要遵循以下步骤即可：

（1）发送命令，即对 NAND FLASH 采取哪种操作，读、写还是擦除？

（2）发送地址，即对 NAND FLASH 的哪一页进行上述操作？

（3）发送数据，在此期间要检测 NAND FLASH 的内部状态。

NAND FLASH 支持的命令如表 12-3 所示，发送命令时只需要将命令字写入命令寄存器即可。

表 12-3　NAND FLASH 支持的命令

功　能	第 1 个访问周期	第 2 个访问周期
读命令	00H	30H
读 ID 命令	90H	
复位命令	FFH	
写页	80H	10H
块擦除	60H	D0H
随机数输入	85H	
随机数输出	05H	E0H
读状态	70H	

NAND FLASH 内部的地址如何确定呢？从 K9F2G08U0B 数据手册上可以找到有关地址序列的信息，如图 12-5 所示。

	I/O 0	I/O 1	I/O 2	I/O 3	I/O 4	I/O5	I/O6	I/O7	
1st Cycle	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	Column Address
2nd Cycle	A_8	A_9	A_{10}	A_{11}	*L	*L	*L	*L	Column Address
3rd Cycle	A_{12}	A_{13}	A_{14}	A_{15}	A_{16}	A_{17}	A_{18}	A_{19}	Row Address
4th Cycle	A_{20}	A_{21}	A_{22}	A_{23}	A_{24}	A_{25}	A_{26}	A_{27}	Row Address
5th Cycle	A_{28}	*L	*L	*L	*L	*L	*L	*L	Row Address

图 12-5　NAND FLASH 地址序列

结合图 12-1 可以很容易地理解地址序列的含义。

所谓的列地址即在一页中的地址，因为每页大小是 2 K+64 Bytes，所以需要 12 根地址线来寻址，也即 A0～A11，整个 NAND FLASH 包含 128 K 个页面，则需要 17 根地址线来寻址每一个页面，即 A12～A28，这也就是所谓的行地址。

进一步讲，页面大小是 2 K+64 Bytes，因此页内地址使用 A0～A11 来寻址；每一块包含 64 页，因此使用地址 A12～A17 来寻址，整个 NAND FLASH 包含 2K 个块，所以使用 A18～A28 来寻址。

注意： *初学者阅读到这里，一般情况下是一头雾水，不是说 NAND FLASH 有多难，原因在于刚刚接触到 NAND FLASH，所以感觉比较难理解，请大胆阅读，结合本章实验，慢慢就理解了，等再看第 2 遍的时候，也许本书已经有点小儿科了。*

12.1.4　S3C2440 NAND FLASH 控制器

前文提到 NAND FLASH 控制器能够提供访问 NAND FLASH 所需的时序信息，到底是怎么产生的呢？在学习 SDRAM 初始化时已经给读者展示过初始化 SDRAM，只需要根据 SDRAM 数据手册给出的时序参数配置好 S3C2440 相应的寄存器即可。对 NAND FLASH 控制器的初始化也是同样的道理，只需要根据 K9F2G08U0B 数据手册给出的时序参数，正确初始化 S3C2440 处理器相关的寄存器即可。

结合具体的寄存器，操作 NAND FLASH 的基本步骤可以概括为下述两步。

1．配置 GPACON 寄存器，将 GPA17～GPA22 设置为 NAND FLASH 控制器信号模式

GPACON 寄存器部分位的含义如图 12-6 所示，需要将 GPA17～GPA22 配置为 NAND FLASH 所需要的信号模式。当然，如果不配置，在复位后，虽然系统默认是这种模式，最好还是配置。

可以使用如下代码实现：

```
rGPACON  &=  ~(0X3F << 17);
rGPACON  |=  (0X3F << 17);
```

GPACON	位	描　述	
GPA24	[24]	Reserved	
GPA23	[23]	Reserved	
GPA22	[22]	0=Output	1=nFCE
GPA21	[21]	0=Output	1=nRSTOUT
GPA20	[20]	0=Output	1=nFRE
GPA19	[19]	0=Output	1=nFWE
GPA18	[18]	0=Output	1=ALE
GPA17	[17]	0=Output	1=CLE

图 12-6　GPACON 寄存器部分位的含义

2. 配置 NAND FLASH，主要是初始化寄存器 NFCONF

对该寄存器的初始化主要是确定 TACLS、TWRPH0 和 TWRPH1 三个位的值即可，其他位不需要初始化。

S3C2440 处理器数据手册中给出的上述 3 个值的关系，如图 12-7 所示。

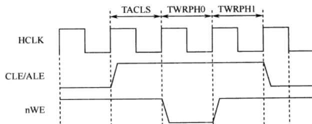


图 12-7　CLE & ALE Timing（TACLS=1，TWRPH0=0，TWRPH1=0）

从图 12-7 中可以看出：

- TACLS 表征了从 CLE/ALE 锁存信号有效到写信号使能经过的时间，具体时间=HCLK 时钟周期×TACLS。
- TWRPH0 表征了写有效的持续时间，具体时间= HCLK 时钟周期×(TWRPH0+1)。
- TWRPH1 表征了写无效到锁存无效之间的时间，具体时间= HCLK 时钟周期×(TWRPH1+1)。

上述 3 个值的初始化需要进一步结合 K9F2G08U0B 数据手册给出的相关信号的时序关系进行讲解。K9F2G08U0B 数据手册给出的相关信号的时序关系如图 12-8 所示。

对比图 12-7 和图 12-8 可以得到下列对应关系：

- TACLS 表征的锁存有效到写有效的时间与 K9F2G08UOB 手册中的 tCLS-tWP 对应。
- TWRPH0 表征的写有效的持续时间与 K9F2G08UOB 手册中的 tWP 对应。
- TWRPH1 表征的写无效到锁存无效之间的时间与 K9F2G08UOB 手册中的 tCLH 对应。

因此只要确定了 tCLS、tWP 和 tCLH 三个参数，则 TACLS、TWRPH0 和 TWRPH1 三个参数的值即可确定。

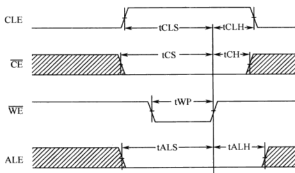


图 12-8 K9F2G08U0B 数据手册给出的相关信号的时序关系

这时需要进一步结合具体的时序参数进行计算，K9F2G08U0B 数据手册给出的相关信号的时序参数如表 12-4 所示。

表 12-4 相关信号的时序参数

Parameter	Symbol	Min	Max	Unit
CLE Setup Time	tCLS(1)	12	-	ns
CLE Hold Time	tCLH	5	-	ns
CE Setup Time	tCS(1)	20	-	ns
CE Hold Time	tCH	5	-	ns
WE Puise Width	tWP	12	-	ns
ALE Setup Time	tALS(1)	12	-	ns
ALE Hold Time	tALH	5	-	ns
Data Setup Time	tDS(1)	12	-	ns
Data Hold Time	tDH	5	-	ns
Write Cycle Time	tWC	25	-	ns
WE High Hold Time	tWH	10	-	ns
Address to Data Loading Time	tADL(2)	100	-	ns

可见，tCLS=12 ns、tWP=12 ns、tCLH=5 ns，当然这只是一个参考，在具体初始化时可以将时间适当延长一点。

因为在系统初始化阶段，HCLK=100 MHz。因此，HCLK 的时钟周期为 1/100 M=10 ns。

由 tWP=12 ns，TWRPH0=4 即可，进而 TACLS =1，TWRPH1=0 就可以满足要求。

初始化 NAND FLASH 过程中涉及下述主要寄存器。

- 寄存器 NFCONT，用于开启 NAND FLASH 控制器。
- 向寄存器 NFCMD 写入命令。

向 NAND FLASH 发送命令，只需要将命令写入该寄存器即可，NAND FLASH 控制器会根据上述参数自动产生出访问 NAND FLASH 所需的命令信号。

例：向 NAND FLASH 发送命令。

```
#define NF_CMD(cmd)            {rNFCMD   = (cmd); }
```

- 向寄存器 NFADDR 写入地址。

将要访问的地址写入到这个寄存器，NAND FLASH 控制器会根据上述参数自动产生出访问 NAND FLASH 所需的地址信号。

例：向 NAND FLASH 发送地址。

```
#define NF_ADDR(addr)          {rNFADDR = (addr); }
```

- 使用寄存器 NFDATA 进行数据读写，在此期间需要检查 NFSTAT 来检测 NAND FLASH 的状态。
- 寄存器 NFSTAT，用于指示 NAND FLASH 是否处于忙状态。

例：检测 NAND FLASH 是否处于忙状态。

```
#define NF_DETECT_RB()         {while(!(rNFSTAT&(1<<2)));}
```

12.1.5 使用宏代替简单的函数

在程序开发过程中，经常将一个很大的工程分解为几个小的模块，每个模块使用单独的函数来实现，最后在工程中通过对各个模块函数的调用来实现整个工程所完成的功能，这也是典型的模块化开发技巧。但是，当项目中的函数调用关系复杂时，尤其是存在多级函数调用时，需要将每一级的返回地址保存在栈中，容易导致栈溢出，此外函数的调用开销也会逐渐加大。

为了更好地解决上述问题，一般使用宏的形式来实现规模较小的函数。因为宏调用是在预处理阶段，由预处理器对源程序中的宏进行展开，所以宏展开不占用运行时间。因为每一次宏调用都需要进行宏展开，所以会加大程序的代码量，因此规模较大的函数不宜使用宏的形式来实现。

下面结合 NAND FLASH 操作函数，向读者展示宏的使用。为了讲述方便，将 2440addr.h 文件中关于 NAND FLASH 控制器有关寄存器引用如下：

```
#define rNFCONT          (*(volatile unsigned *)0x4E000004)
#define rNFCMD           (*(volatile unsigned *)0x4E000008)
#define rNFADDR          (*(volatile unsigned *)0x4E00000C)
#define rNFDATA          (*(volatile unsigned *)0x4E000010)
#define rNFDATA8         (*(volatile unsigned char *)0x4E000010)
```

下面是具体的 NAND FLASH 操作函数：

- #define NF_Enable()　　{rNFCONT　&=　~(1<<1); }

前文讲到，NFCONT 的第 1 位用于控制 NAND FLASH 的片选信号，当该位为 0 时，片选信号使能，可以对 NAND FLASH 进行读、写、擦除等操作。

该宏使用按位与运算将 NFCONT 寄存器的第 1 位清零，使能 NAND FLASH。

- #define NF_Disable()　　{rNFCONT |= (1<<1); }

NFCONT 的第 1 位用于控制 NAND FLASH 的片选信号，当该位为 1 时，片选信号无效，此时，无法对 NAND FLASH 进行读、写、擦除等操作。

该宏使用按位或运算将 NFCONT 寄存器的第 1 位置 1，使 NAND FLASH 无效。

- #define NF_Send_Cmd(cmd) {rNFCMD = (cmd); }

该宏实现的功能是：向 NAND FLASH 发送命令。

只需要将所要发送的命令写入命令寄存器 NFCMD 即可，NAND FLASH 控制器会自动按照 NAND FLASH 操作时序将该命令发送到 NAND FLASH。

- #define NF_Send_Addr(addr) {rNFADDR = (addr); }

该宏实现的功能是：向 NAND FLASH 发送地址。

只需要将所要发送的地址写入地址寄存器 NFADDR 即可，NAND FLASH 控制器会自动按照 NAND FLASH 操作时序以该地址对 NAND FLASH 进行寻址。

- #define NF_Send_Data(data) {rNFDATA8 = (data); }

该宏实现的功能是：向 NAND FLASH 发送命令。

只需要将所要发送的命令写入数据寄存器即可，NAND FLASH 控制器会自动按照 NAND FLASH 操作时序将该数据发送到 NAND FLASH。

注意：这里的数据寄存器是这么定义的：

```
#define rNFDATA8(*(volatile unsigned char *)0x4E000010)
```

为什么这么定义呢？请读者参见如图 12-2 所示的 NAND FLASH 接口电路，注意到数据线是 IO0～IO7，只有 8 根数据线，因此每次只能发送一个字节的数据，只需要使用数据寄存器 NFDATA 的低 8 位即可。为了形象地展示上述方法的来源，笔者找到了 S3C2440 处理器的数据手册，如图 12-9 所示，可见，确实只使用了低 8 位，这也是提醒初学者，很多知识点都是从数据手册去挖掘，当然，需要初学者有耐心，善于思考问题。

A.Byte Access

Register	Endian	bit[31:24]	bit[23:16]	bit[15:8]	bit[7:0]
NFDATA	Little/Big	Invalid value	Invalid value	Invalid value	1stI/O[7:0]

图 12-9　8-bit NAND FLASH 数据寄存器使用情况

- #define NF_Enable_RB() {rNFSTAT |= (1<<2); }

这个宏定义非常关键，这条宏定义的功能是开启忙信号检测功能。开启该功能后，可以通过读取寄存器 NFSTAT 的第 0 位来观测 NAND FLASH 是否处于忙状态。如果该位为 0，则说明 NAND FLASH 内部操作正在进行，处于忙状态。如果该位为 1，则说明 NAND FLASH 内部操作已经完成，可以对其进行后续的操作，即处于空闲状态。

在对 NAND FLASH 操作时需要检测 NAND FLASH 是否处于忙状态，当然 NAND FLASH 有专门的忙信号检测命令，可以使用读状态命令（0x70H）来读取 NAND FLASH 的内部状态。

有很多初学者也这么使用。但是，为什么使用 NAND FLASH 控制器呢？其实就是为了便于操作和控制 NAND FLASH。因此最简便的方法就是使用 NAND FLASH 提供的忙信号检测功能，只需要读取该位的值即可确定 NAND FLASH 是否处于忙状态，而不是使用命令去查询。

- #define NF_Check_Busy(){while(!(rNFSTAT&(1<<2)));}

该宏实现的功能是：检测 NAND FLASH 是否处于忙状态。

当 NFSTAT 的第 2 位为 0 时，说明 NAND FLASH 内部操作正在进行，处于忙状态。

此时，rNFSTAT&(1<<2)为 0，则!(rNFSTAT&(1<<2))为 1，while 语句条件为真，则一直执行空等待。

当该位置 1 时，说明 NAND FLASH 处于空闲状态。此时，rNFSTAT&(1<<2)为 1，则!(rNFSTAT&(1<<2))为 0，while 语句条件为假，则结束 while 循环。

- #define NF_Read_Byte() (rNFDATA8)

该宏定义仅仅是为了使用起来方便和便于记忆，没有实质性的含义。

例：可以使用如下代码从 NAND FLASH 读取一个字节。

```
char   pMID ;
pMID = NF_Read_Byte();
```

12.2 NAND FLASH 基础实验

前文讲到的几个宏主要是实现了启动/关闭 NAND FLASH、检测忙信号和从 NAND FLASH 读取一个字节数据的功能。下面将上面几个宏进行综合调用就可以实现基本的 NAND FLASH 驱动程序。

12.2.1 NAND FLASH 基本操作函数分析

下面重点讲解 NAND FLASH 基本操作函数，其中包括复位、初始化、页写入、页读取、块擦除等操作函数的实现原理及具体程序。

这里再讲述一个小技巧：一般使用命令时，不是一遍一遍地查阅 NAND FLASH 数据手册找到相应的命令，而是将所有的命令以宏的形式定义好，以后使用的时候直接使用相应的宏即可。表 12-3 展示了 NAND FLASH 支持的部分命令，下面以宏的形式给出各条命令的定义。

```
#define CMD_READ1          0x00     //  Read1
#define CMD_READ2          0x30     //  Read2
#define CMD_READID         0x90     //  ReadID
#define CMD_WRITE1         0x80     //  Write phase 1
#define CMD_WRITE2         0x10     //  Write phase 2
#define CMD_ERASE1         0x60     //  Erase phase 1
#define CMD_ERASE2         0xd0     //  Erase phase 2
#define CMD_STATUS         0x70     //  Status read
#define CMD_RESET          0xff     //  Reset
```

- NAND FLASH 复位函数

一般在操作 NAND FLASH 之前需要复位。

基本方法：向 NAND FLASH 命令寄存器写入复位命令即可。

```
static void NF_Reset()
{
1      NF_Enable();
```

```
2     NF_Enable_RB();
3     NF_Send_Cmd(CMD_RESET);
4     NF_Check_Busy();
5     NF_Disable();
}
```

第 1 行，调用宏 NF_Enable()，打开 NAND FLASH 片选信号，选中 NAND FLASH。

第 2 行，开启忙信号检测功能，开启之后，以后就可以通过检测寄存器 NFSTAT 的第 2 位来确定 NAND FLASH 是否处于忙状态。

第 3 行，向 NAND FLASH 发送复位命令，只需要将复位命令写入命令寄存器，NAND FLASH 控制器就会将该命令发送给 NAND FLASH，NAND FLASH 收到该命令后会自动执行复位操作，读者不需要关心。

第 4 行，检测忙信号，第 3 行刚发送了复位命令，NAND FLASH 接收到该命令后会自动执行复位操作，这些不需要读者关心，但是读者关心的是：复位操作什么时候执行完呢？

在 NAND FLASH 执行命令期间，忙信号一直是低电平；当执行完相应的命令后，忙信号会自动变为高电平。因此，只需要检测忙信号的电平就可以确定命令是否已经执行完毕。

第 5 行，关闭片选信号，一般使用 NAND FLASH 时就打开片选信号，使用完后就关闭，这种情况主要是为了省电。

- NAND FLASH 初始化函数

NAND FLASH 初始化就是将相应的时序参数写入 NAND FLASH 控制寄存器即可。前文分析中已经得到了 TACLS、TWRPH0、TWRPH1 的值。

```
#define TACLS            1
#define TWRPH0           4
#define TWRPH1           0
void NF_Init(void)
{
1     rGPACON     &=   ~(0X3F << 17) ;
2     rGPACON     |=   (0X3F << 17) ;
3     rNFCONF     =    (TACLS<<12)|(TWRPH0<<8)|(TWRPH1<<4);
4     rNFCONT     =    (0<<12) |(1<<0);
5     rNFSTAT     =    0;
6     NF_Reset();
}
```

第 1～2 行，配置 GPACON 寄存器，将 GPA17～GPA22 设置为 NAND FLASH 控制器信号输出模式。

第 3 行，将上述三个参数写入寄存器 NFCONF 的相应位。

第 4 行，寄存器 NFCONT 的第 0 位控制 NAND FLASH 的开启与关闭，第 12 位与 NAND FLASH 的加密有关，初学者可以暂时不考虑这个地方，先按照这种方法初始化，接触 NAND FLASH 时间长了，读者再自行学习这方面的知识即可。

第 5 行，将 NFSTAT 寄存器清零，这个寄存器中保存了与 NAND FLASH 控制有关的

状态位，在初始化阶段需要将其全部清零。

第 6 行，配置完相应的寄存器后，只需要调用 NAND FLASH 复位函数就可实现对 NAND FLASH 的复位操作，以后就可以对 NAND FLASH 进行读、写和擦除等操作了。

- NAND FLASH 页写入函数

NAND FLASH 主要用于大量数据段的存取，因此向 NAND FLASH 写入数据是以页为单位的。当然，有些类型的 NAND FLASH 也支持一个字节一个字节地写入（这就是所谓的随机读写，K9F2G08U0B 就支持单字节写入，这部分知识在 NAND FLASH 高级实验部分讲解）。

NAND FLASH 页写入函数的功能是向某一页写入数据。前文讲到，NAND FLASH 由不同的块组成，每块包含 64 页，因此，要想写入某一页，首先需要确定该页属于哪一个块。

结合前面小区和 NAND FLASH 的类比，向某一页写入数据，就像给某一层楼的所有用户送快递，首先要找到那栋楼，然后再确定那层楼。

NAND FLASH 页写入命令有两个，分别是 0x80 和 0x10。页写入操作的流程为：

（1）发送页写入命令 0x80。

（2）发送页地址。

（3）发送要写入的数据。

（4）发送页写入确认命令 0x10。

（5）检测忙信号。

注意： *为什么需要发送两次命令呢？这是初学者遇到的常见问题。*

当发送页写入确认命令 0x10 之前，并没有将数据写入 NAND FLASH 的存储单元，仅仅是将数据写入了 NAND FLASH 的数据寄存器（Data Register & S/A）里，如图 12-10 所示，只有收到页写入确认命令 0x10 后，NAND FLASH 才会自动将数据寄存器中的数据写入对应的存储单元中。这样做的主要目的是为了降低页写入时间。

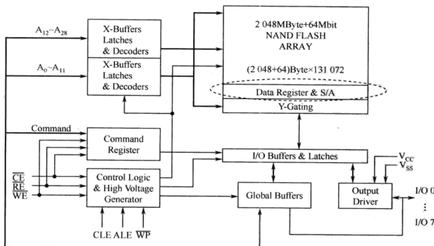


图 12-10　NAND FLASH 系统框图

在发送页写入确认命令之后，NAND FLASH 会自动将数据寄存器中的数据写入对应的存储单元中，这需要一定的时间。因此，在发送完页写入确认命令 0x10 之后，需要不断地检测忙信号，只有 NAND FLASH 真正写完数据后，才能对其进行后续的操作。

NAND FLASH 页写入函数 NF_WritePage()：该函数有三个参数，第 1 个是块号，第 2 个是在该块内的页号，第 3 个是一个指针，指向将要写入的数据的基地址。可以使用如下方式调用：

```
unsigned char buf[2048] ;
NF_WritePage(17,4, buf) ;
```

即将 buf 中的数据写入到 NAND FLASH 的第 17 块中的第 4 页。

这里需要注意一个问题：向地址寄存器中写入的地址指的是 NAND FLASH 中的页的绝对地址，即该页距离第 0 页的绝对地址。由于 NAND FLASH 中含有的页数目太多，为了便于管理，才使用了块的概念，把整个 NAND FLASH 分成了 2K 个块，每块有 64 个页。

因此，上例中第 17 块中的第 4 页的绝对地址为：17×64+4 =1 092 页。

```
void NF_WritePage(unsigned int block,unsigned int page, unsigned char *buffer)
{
1     unsigned int i;
2     unsigned int blockPage = (block<<6)+page;
3     unsigned char *bufPt = buffer;
4     NF_Reset();
5     NF_Enable();
6     NF_Enable_RB();          //开启 RnB 监视模式

7     NF_Send_Cmd(CMD_WRITE1); /*  发送页写入命令  */
8     NF_Send_Addr(0x00);
9     NF_Send_Addr(0x00);
10    NF_Send_Addr((blockPage) & 0xff);
11    NF_Send_Addr((blockPage >> 8) & 0xff);
12    NF_Send_Addr((blockPage >> 16) & 0x1);

13    for(i=0;i<2048;i++)
      {
14        NF_Send_Data(*bufPt++);
      }
15    NF_Send_Cmd(CMD_WRITE2);
16    NF_Check_Busy();

17    NF_Disable();
}
```

第 2 行，计算页的绝对地址，注意这里的小技巧，页的绝对地址=块号×64+页号，但是乘法使用了移位运算来实现，左移 6 位就相当于乘以 64。对于嵌入式系统而言，应尽量

使用移位运算代替乘法运算，毕竟乘法运算需要耗费较长的处理器时间，当然这也是初学者在开发过程中容易忽略的问题之一。

第 4～6 行，复位 NAND FLASH，然后打开 NAND FLASH（在复位结束后关闭了 NAND FLASH，因此需要重新打开），同时开启忙信号检测功能，以后就可以对 NAND FLASH 进行操作，然后通过检测忙信号来获取 NAND FLASH 内部的工作状态。

第 7 行，发送页写入命令。

第 8～12 行，发送页的绝对地址，为什么需要发送 5 次呢？前文讲到 NAND FLASH 的接口电路，地址线和数据线是复用的，接口位宽为 8 位，因此每次只能发送一个字节，又因为 NAND FLASH 的地址是 28 位的，如图 12-11 所示，需要 5 个地址周期才能将地址发送完毕。发送完地址后，NAND FLASH 内部的地址译码电路会自动将收到的地址进行组合，不需要读者关心，但是需要注意发送的顺序，按照先发送低地址，再发送高地址的顺序发送。

	I/O 0	I/O 1	I/O 2	I/O 3	I/O 4	I/O 5	I/O 6	I/O 7	
1st Cycle	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	Column Address
2nd Cycle	A_8	A_9	A_{10}	A_{11}	*L	*L	*L	*L	Column Address
3rd Cycle	A_{12}	A_{13}	A_{14}	A_{15}	A_{16}	A_{17}	A_{18}	A_{19}	Row Address
4th Cycle	A_{20}	A_{21}	A_{22}	A_{23}	A_{24}	A_{25}	A_{26}	A_{27}	Row Address
5th Cycle	A_{28}	*L	*L	*L	*L	*L	*L	*L	Row Address

图 12-11 NAND FLASH 地址周期

前文讲到，地址的低 12 位用于页内寻址，这是对整页进行写入。因此，低 12 位设为 0 即可，如第 8、9 行。因为每次需要发送一个字节，所以要对地址进行恰当的屏蔽。

第 10 行，此时处于第 3 个地址周期，因此需要发送 blockPage（此时 blockPage 是页的绝对地址）的 A12～A19 位，因此，将 blockPage 与 0xff 相与即可。

第 11 行，此时处于第 4 个地址周期，因此需要发送 blockPage（此时 blockPage 是页的绝对地址）的 A20～A27 位。因此，将 blockPage 右移 8 位，然后再与 0xff 相与即可。

第 12 行，此时处于第 5 个地址周期，因此需要发送 blockPage（此时 blockPage 是页的绝对地址）的 A28 位，从图 12-11 可看到，此时虽然是发送 8 位，但是只有第 0 位是有效的，其他位无效。因此，将 blockPage 右移 16 位，然后再与 0x1 相与，只保留第 0 位。

前面已经将地址发送给了 NAND FLASH，NAND FLASH 会自动将上述地址组合，然后译码选中需要写入的页，读者不必关心这些事情。

第 13～14 行，使用 for 循环，将 2 K 字节的数据发送给 NAND FLASH，发送数据是一个字节一个字节地发送，只需要将数据写入数据寄存器，NAND FLASH 控制器会自动将数据发送给 NAND FLASH。

前文讲到，NAND FLASH 此时只是将接收到的数据存入了内部的数据寄存器中，并没有写入相应的页内。

第 15 行，写完数据后，发送页写入确认命令，也就是通知 NAND FLASH 数据发送已经完毕，NANF FLASH 收到该命令后，会自动将数据寄存器中的数据写入由地址译码电路

所选中的地址单元中。

第 16 行，检测忙信号，等待 NAND FLASH 将所有数据写入完毕，在此期间无法对 NNAD FLASH 进行其他操作。

第 17 行，操作完成后，关闭片选信号即可。

- NAND FLASH 页读取函数

前文讲到当发送页写入命令 0x80 后，NAND FLASH 将数据写入了 NAND FLASH 的数据寄存器（Data Register & S/A）里，只有收到页写入确认命令 0x10 后，NAND FLASH 才会自动将数据寄存器中的数据写入对应的存储单元中。

现在讲解 NAND FLASH 页读取函数将变得很简单，页读取操作也需要两个命令：页读取发起命令 0x00 和页读取确认命令 0x30。

基本操作原理：当发送页读取发起命令 0x00 后，需要紧接着发送需要读取的页的绝对地址，然后发送页读取确认命令 0x30，NAND FLASH 收到第 2 个命令 0x30 后，自动将数据从内部存储单元复制到了 NAND FLASH 的数据寄存器（Data Register & S/A）里，在此期间需要检测忙信号（如果忙信号有效，说明 NAND FLASH 存储单元中的数据还没有全部复制到数据寄存器），数据复制完毕后，可以通过读取 S3C2440 内部的特殊功能寄存器 NFDATA 来得到所需要的数据。

NAND FLASH 页读取命令有两个，分别是 0x00 和 0x30。页读取操作的流程为：

（1）发送页读取发起命令 0x00。

（2）发送页地址。

（3）发送页读取确认命令 0x30。

（4）检测忙信号。

（5）从 S3C2440 处理器寄存器 NFDATA 中读取数据。

```
void NF_ReadPage(unsigned int block,unsigned int page, unsigned char * dstaddr)
{
1    unsigned int i;
2    unsigned int blockPage = (block<<6)+page;
3    NF_Reset();
4    NF_Enable();
5    NF_Enable_RB();

6    NF_Send_Cmd(CMD_READ1);          //CMD_READ1= 0x00

7    NF_Send_Addr(0x00);
8    NF_Send_Addr(0x00);
9    NF_Send_Addr((blockPage) & 0xff);
10   NF_Send_Addr((blockPage >> 8) & 0xff);
11   NF_Send_Addr((blockPage >> 16) & 0x1);

12   NF_Send_Cmd(CMD_READ2);     //CMD_READ12= 0x30
```

```
13  NF_Check_Busy();

14  for (i = 0; i < 2048; i++)
    {
15      dstaddr[i] =  NF_Read_Byte();
    }

16  NF_Disable();
}
```

第 2 行，计算页的绝对地址。

第 3～5 行，复位 NAND FLASH，然后打开 NAND FLASH（在复位结束后关闭了 NAND FLASH，因此需要重新打开），同时开启忙信号检测功能，以后就可以对 NAND FLASH 进行操作，然后通过检测忙信号来获取 NAND FLASH 内部的工作状态。

第 6 行，发送页读取发起命令 0x00。

第 7～11 行，发送页的绝对地址。

第 12 行，发送页读取确认命令 0x30。

第 13 行，检测忙信号，等待 NAND FLASH 将内部存储单元中的数据复制到数据寄存器中。

第 14～15 行，使用 for 循环，从 NFDATA 寄存器读取数据即可。

第 16 行，操作完成后，关闭片选信号即可。

- NAND FLASH 块擦除函数

前文讲解了 NAND FLASH 页读、写函数，但是对 NAND FLASH 写之前，需要先进行擦除，这是由 NAND FLASH 自身的存储器结构决定的。

对某一个存储单元来说，只能向该单元写 0，无法向其写 1。所谓擦除是将所有的存储单元全部写为 1，然后再对其进行写操作，0 可以写入，1 虽然无法写入，但是由于擦除时该存储单元已经为 1，所以该存储单元保留 1。因此，利用擦除操作达到间接地向存储单元写 1 的目的。

擦除操作是以块为单位进行的，无法对一页进行擦除操作。

小提示：结合前面小区和 NAND FLASH 的类比。擦除操作就像在小区里打扫卫生，NAND FLASH 的一个块就像小区里的一栋楼，擦除只能以块为单位进行，就像打扫卫生时，只能整栋楼都打扫卫生。

块擦除基本原理：发出块擦除发起命令 0x60 后，需要发送三个地址周期的块地址（注意，因为块地址只使用 A12～A28，所以需要三个地址周期），最后发送块擦除确认命令（D0H），NAND FLASH 接收到块擦除确认命令后会启动内部的擦除过程。可以通过检测忙信号来确定擦除是否完成。

前文讲到，每个页面大小是 2 K+64 Bytes，因此，页内地址使用 A0～A11 来寻址；每一块包含 64 页，因此，使用地址 A12～A17 来寻址，整个 NAND FLASH 包含 2K 个块，所以使用 A18～A28 来寻址，如图 12-12 所示。

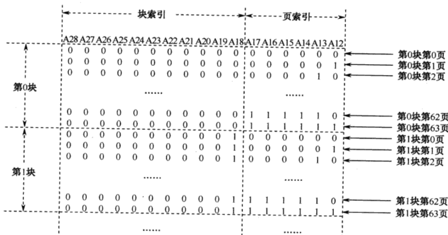


图 12-12 块索引和页索引

块擦除操作需要两个命令：块擦除发起命令 0x60 和块擦除确认命令 0xD0。块擦除操作的流程为：

（1）块擦除发起命令 0x60。

（2）发送块地址。

（3）块擦除确认命令 0xD0。

（4）检测忙信号。

```
int NF_EraseBlock(unsigned int block)
{
1      unsigned int blockNum = block<<6 ;
2      NF_Reset();
3      NF_Enable();
4      NF_Enable_RB();

5      NF_Send_Cmd(CMD_ERASE1);

6      NF_Send_Addr(blockNum & 0xff);
7      NF_Send_Addr((blockNum >>8) & 0xff);
8      NF_Send_Addr((blockNum >>16) & 0xff);

9      NF_Send_Cmd(CMD_ERASE2) ;

10     NF_Check_Busy() ;

11     NF_Disable() ;
       return 1 ;
}
```

第 1 行，将块号左移 6 位，因为该地址周期发送的是 A12～A18，但是 A12～A17 是索引该块内的页面，如图 12-12 所示，块擦除操作将该块内的所有页面均擦除。

第 2～4 行，复位 NAND FLASH，然后打开 NAND FLASH（在复位结束后关闭了 NAND FLASH，因此需要重新打开），同时开启忙信号检测功能，以后就可以对 NAND FLASH 进行操作，然后通过检测忙信号来获取 NAND FLASH 内部的工作状态。

第 5 行，发送块擦除发起命令 0x60。

第 6～8 行，发送块的绝对地址。

第 9 行，发送块擦除确认命令 0xD0，NAND FLASH 收到该命令后，会自动执行块擦除操作。

第 10 行，检测忙信号，等待 NAND FLASH 擦除完毕。

第 11 行，操作完成后，关闭片选信号即可。.

12.2.2 NAND FLASH 基础实验之页读写

经过前面的讲解，对 NAND FLASH 的复位、初始化、页读写和块擦除有了基本的了解，下面结合具体的实验，向读者展示具体如何使用这些操作。

NAND FLASH 基础实验基本功能：利用页写入函数向第 17 块的第 4 页写入数据，然后使用页读取函数将写入的数据读出，打印到串口上。

NAND FLASH 实验文件布局如图 12-13 所示，其中需要使用启动代码中的 nand.h 和 nand.c 文件。

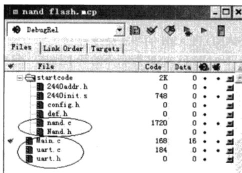


图 12-13　NAND FLASH 实验文件布局

nand.h 文件内容如下：

```
#ifndef __NAND_H
#define __NAND_H

1    #define CMD_READ1        0x00     //  Read1
2    #define CMD_READ2        0x30     //  Read2
3    #define CMD_READID       0x90     //  ReadID
4    #define CMD_WRITE1       0x80     //  Write phase 1
5    #define CMD_WRITE2       0x10     //  Write phase 2
```

```
6    #define CMD_ERASE1          0x60        //  Erase phase 1
7    #define CMD_ERASE2          0xd0        //  Erase phase 2
8    #define CMD_STATUS          0x70        //  Status read
9    #define CMD_RESET           0xff        //  Reset

10   #define NF_Send_Cmd(cmd)        {rNFCMD   = (cmd); }
11   #define NF_Send_Addr(addr)      {rNFADDR = (addr); }
12   #define NF_Send_Data(data)      {rNFDATA8 = (data); }
13   #define NF_Enable()             {rNFCONT &= ~(1<<1); }
14   #define NF_Disable()            {rNFCONT |= (1<<1); }
15   #define NF_Enable_RB()          {rNFSTAT |= (1<<2); }    //开启 RnB 监视模式;
16   #define NF_Check_Busy()         {while(!(rNFSTAT&(1<<2)));}
17   #define NF_Read_Byte()          (rNFDATA8)
18   #define TACLS                   1
19   #define TWRPH0                  4
20   #define TWRPH1                  0

21   extern void NF_Init(void) ;
22   extern void NF_ReadPage(unsigned int block,unsigned int page, unsigned char * dstaddr) ;
23   extern void NF_WritePage(unsigned int block,unsigned int page, unsigned char *buffer) ;
24   extern int  NF_EraseBlock(unsigned int block) ;//

#endif
```

第 1～9 行，定义了 NAND FLASH 的基本命令。

第 10～17 行，定义了 NAND FLASH 控制器的基本操作，包括 NAND FLASH 的开启关闭，检测忙信号，发送命令、地址、数据等操作。

注意：rNFDATA8 是在 2440addr.h 文件中定义的，定义如下。

#define rNFDATA8(*(volatile unsigned char *)0x4E000010)

第 18～20 行，定义了 NAND FLASH 的 3 个时序参数，具体分析见 12.1.4 节。

第 21～24 行，使用 extern 关键字声明了 4 个外部函数，这样就可以在其他文件中使用这几个函数。

nand.c 文件内容如下：

```
#include "2440addr.h"
#include "Nand.h"

static void NF_Reset()
{
    NF_Enable();
    NF_Enable_RB();
    NF_Send_Cmd(CMD_RESET);
    NF_Check_Busy();
```

```
        NF_Disable();
}

void NF_Init(void)
{
        rGPACON &= ~(0X3F << 17) ;
        rGPACON |= (0X3F << 17) ;
        rNFCONF = (TACLS<<12)|(TWRPH0<<8)|(TWRPH1<<4);
        rNFCONT = (0<<12)|(1<<0);
        rNFSTAT = 0;
        NF_Reset();
}

void NF_ReadPage(unsigned int block,unsigned int page, unsigned char * dstaddr)
 {
        unsigned int i;
unsigned int blockPage = (block<<6)+page;

        NF_Reset();
        NF_Enable();
        NF_Enable_RB();

        NF_Send_Cmd(CMD_READ1);          //CMD_READ1= 0x00

        NF_Send_Addr(0x00);
        NF_Send_Addr(0x00);
        NF_Send_Addr((blockPage) & 0xff);
        NF_Send_Addr((blockPage >> 8) & 0xff);
        NF_Send_Addr((blockPage >> 16) & 0x1);

        NF_Send_Cmd(CMD_READ2);     //CMD_READ12= 0x30

        NF_Check_Busy();

        for (i = 0; i < 2048; i++)
        {
              dstaddr[i] =   NF_Read_Byte();
        }

        NF_Disable();
 }

void NF_WritePage(unsigned int block,unsigned int page, unsigned char *buffer)
```

```
{
        unsigned int i;
        unsigned int blockPage = (block<<6)+page;
        unsigned char *bufPt = buffer;

        NF_Reset();
        NF_Enable();    //控制器使能
        NF_Enable_RB(); //开启 RnB 监视模式

        NF_Send_Cmd(CMD_WRITE1); /*  写第 1 条命令  */

        NF_Send_Addr(0x00);
        NF_Send_Addr(0x00);
        NF_Send_Addr((blockPage) & 0xff);
        NF_Send_Addr((blockPage >> 8) & 0xff);
        NF_Send_Addr((blockPage >> 16) & 0x1);
        for(i=0;i<2048;i++)
        {
              NF_Send_Data(*bufPt++);
        }

        NF_Send_Cmd(CMD_WRITE2);
        NF_Check_Busy();
        NF_Disable();
}
int NF_EraseBlock(unsigned int block)
{
        unsigned int blocknum=(block<<6);
        NF_Reset();
        NF_Enable();
        NF_Enable_RB();

        NF_Send_Cmd(CMD_ERASE1);

        NF_Send_Addr( blocknum & 0xff);
        NF_Send_Addr((blocknum>>8) & 0xff);
        NF_Send_Addr((blocknum>>16) & 0xff);

        NF_Send_Cmd(CMD_ERASE2) ;

        NF_Check_Busy() ;

        NF_Disable() ;
```

```
        return 1 ;
    }
```

主要实现了 nand.h 文件中声明的几个函数，对这几个函数在前面已经进行了讲解。

UART 模块包含两个文件：uart.h 和 uart.c 文件。

uart.h 文件中声明了 UART 初始化函数 Uart0_Init ()。

```
#ifndef __UART_H__
#define __UART_H__


extern void Uart0_Init(unsigned int baudrate) ;
extern void putc(unsigned char c) ;
extern unsigned char getc(void) ;

#endif
```

uart.c 文件对上述 3 个函数进行了具体的实现。

```
#define PCLK 50000000                    //时钟源设为 PCLK
void Uart0_Init(unsigned int baudrate)
{
1       rGPHCON &= ~((3 << 4) | (3 << 6)) ;
2       rGPHCON |= ((2 << 4) | (2 << 6)) ; //GPH2--TXD[0];GPH3--RXD[0]
3       rGPHUP   = 0x00 ;
4       rULCON0 |= 0x03 ;                       //8 个数据位，1 个停止位
5       rUCON0   = 0x05 ;
6       rUBRDIV0 = (int) (PCLK / baudrate / 16)- 1;
7       rURXH0   = 0;
}
```

第 1～3 行，将 GPH2、GPH3 配置为 TXD、RXD 模式。

第 4 行，设置寄存器 ULCON0，设置数据发送格式为：8 个数据位，1 个停止位，无校验位。

第 5 行，发送模式和接收模式都使用查询方式。

第 6 行，设置波特率，其中波特率作为一个参数传递到该初始化函数。

第 7 行，将 URXH0 清零。

```
void putc(unsigned char c)
{
        rUTXH0 = c ;
        while(!(rUTRSTAT0 & (1 << 2))) ;//等待上个字符发送完毕
}
```

该函数可以发送一个字符。

首先将要发送的字符存入寄存器 UTXH0 中，然后等待发送完毕，发送完毕后，

UTRSTAT0 寄存器的第 2 位会置 1，然后跳出 while 循环。

```
unsigned char getc(void)
{
      unsigned char c ;
      while(!(rUTRSTAT0 & (1 << 0))) ;
      c = rURXH0 ;
      return c ;
}
```

该函数实现接收一个字符，首先是等待接收完毕，接收完毕后，UTRSTAT0 的第 0 位置 1，跳出 while 循环，将 URXH0 中的值读出即可。

Main.c 文件内容如下：

```
#include "config.h"
#include "uart.h"
#include "nand.h"

1     unsigned char table[]={0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,
                              0x38,0x39,0x41,0x42,0x43,0x44,0x45,0x46} ;
2     void IO_Init() ;
      int Main()
{
3     unsigned char srcbuf[2048],dstbuf[2048] ;
      unsigned int i ;
4     IO_Init() ;
5     for(i = 0 ; i < 2048 ; i++)
      {
6           srcbuf[i] = i ;
      }
7     NF_WritePage(17,4,srcbuf) ;
8     NF_ReadPage(17,4,dstbuf) ;
      while(1)
      {

9           for(i = 0 ; i < 2048 ; i++)
            {
10                putc(table[dstbuf[i]/16]) ;
11                putc(table[dstbuf[i]%16]) ;
12                putc('   ') ;
            }

      }
      return 1 ;
}
```

```
13    void IO_Init()
      {
14        Uart0_Init(115200);
15        NF_Init();
      }
```

第 1 行，定义了一个内存表，里面存放是的 0～F 的 ASCII 码，主要是为了向串口发送数据方便。例如，想让串口调试助手显示数字 0，只需要发送 0 的 ASCII 码 0x30 即可。

第 2 行对 IO_Init()函数进行了声明，第 13 行对该函数进行了定义（即具体实现）。因为在第 4 行要调用该函数，所以当函数调用发生在函数定义之前时需要提前对函数声明，否则编译器报错。

第 5～6 行，使用 for 循环对 srcbuf 数字进行了赋值，由于 srcbuf 的每个存储单元是一个字节，所以 srcbuf 能存储的最大数据是 0xFF，所以经过赋值后，srcbuf 数组中的数据为 0x00，0x01，0x02，…，0xFF，0x00，0x01，0x02，…，0xFF，依此类推。

第 7 行，调用 NAND FLASH 页写入函数 NF_WritePage()，将 srcbuf 中的数据写入第 17 块第 4 页。

第 8 行，调用 NAND FLASH 页读取函数 NF_ReadPage()，将第 17 块第 4 页中的数据读出，存储到 dstbuf 数组中。

第 9～11 行，将读到的数据打印到串口，以十六进制数显示出来。

第 13～15 行，调用了串口和 NAND FLASH 初始化函数。读者可能会问：为什么不将第 14、15 行直接写在 Main()函数里呢？主要是当系统中的模块较多时，每个模块都需要初始化，一般将初始化函数写在一起，主要是看起来比较清晰。

12.2.3 页读写实例测试

将上述工程编译，生成.bin 格式的二进制文件，将其下载到 NAND FLASH 中，打开超级终端，波特率设为 115200，超级终端上显示了程序执行的结果，如图 12-14 所示。

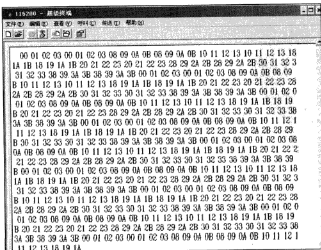


图 12-14 超级终端显示结果（修正前）

为什么读出来的结果不对呢？写进去的是 0x00，0x01，0x02，0x03，0x04，…，0xFF，为什么读出来数据不对呢？问题出在哪里呢？

前文讲到，对 NAND FLASH 每次写之前都需要进行擦除操作。对！问题就是出在这里。因为上面的代码并没有对 NAND FLASH 进行擦除操作，所以导致写入的数据有一部分不正确。因此，在写操作之前需要对第 17 块进行擦除操作，然后才是写入操作。

向 Main.c 文件添加 NAND FLASH 块擦除函数（如下第 7 行），将第 17 块擦除，修改后的 Main.c 文件内容如下：

```
#include "config.h"
#include "uart.h"
#include "nand.h"

1     unsigned char table[]={0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,
                              0x38,0x39,0x41,0x42,0x43,0x44,0x45,0x46} ;
2     void IO_Init() ;
int Main()
{
3         unsigned char srcbuf[2048],dstbuf[2048] ;
      unsigned int i ;
4         IO_Init() ;
5         for(i = 0 ; i < 2048 ; i++)
      {
6             srcbuf[i] = i ;
      }
7         NF_EraseBlock(17) ;
8         NF_WritePage(17,4,srcbuf) ;
9         NF_ReadPage(17,4,dstbuf) ;
      while(1)
      {
          for(i = 0 ; i < 2048 ; i++)
          {
10                putc(table[dstbuf[i]/16]) ;
11                putc(table[dstbuf[i]%16]) ;
12                putc('  ') ;
          }

      }
      return 1 ;
}
13    void IO_Init()
{
14        Uart0_Init(115200) ;
15        NF_Init() ;
}
```

将上述工程编译，生成.bin格式的二进制文件，将其下载到NAND FLASH中，打开超级终端，波特率设为115200，超级终端上显示了程序执行的结果，如图12-15所示。



图12-15 超级终端显示结果（修正后）

可见，从0x00开始递增到0xFF（在图12-15中，0x00和0xFF已经用圆圈标出），一直循环，与前面的分析相吻合。

一般初学者在操作NAND FLASH时，对块擦除没有基本的概念，下面结合擦除后的效果给读者展示一下。

前文讲到，擦除后，存储单元全部变为1。因此不难理解，擦除后，读到的数据应该是0xFF，因为是对第17块进行的擦除，又知道，每一块含有64页，所以第17块的第0～63页里面的内容应该都是0xFF。

下面在前面Main.c文件的基础上，读取第17块第60页的数据（如下第5行）。

修改后，Main.c文件内容如下：

```
#include "config.h"
#include "uart.h"
#include "nand.h"


1    unsigned char table[]={0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,
                                   0x38,0x39,0x41,0x42,0x43,0x44,0x45,0x46} ;
2    void IO_Init() ;
int Main()
{
3        unsigned char srcbuf[2048],dstbuf[2048] ;
         unsigned int i ;
4        IO_Init() ;
5        NF_ReadPage(17,60,dstbuf) ;
6        while(1)
7        {
             for(i = 0 ; i < 2048 ; i++)
```

```
                {
8                       putc(table[dstbuf[i]/16]) ;
9                       putc(table[dstbuf[i]%16]) ;
10                      putc('   ') ;
                }


        }
    return 1 ;
}
11  void IO_Init()
{
12          Uart0_Init(115200) ;
13          NF_Init() ;
}
```

将上述工程编译，生成.bin格式的二进制文件，将其下载到NAND FLASH中，打开超级终端，波特率设为115200，超级终端上显示了程序执行的结果，如图12-16所示。

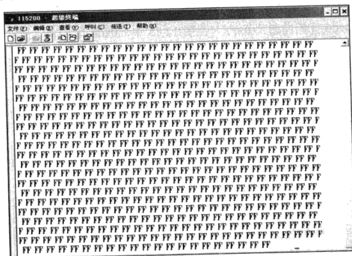


图12-16　测试结果

可见，读取到的数据是0xFF，这说明擦除后，存储单元中的数据全变为1了。

此外，读者也可以尝试着读取第17块中其他的页面，在此不做赘述。

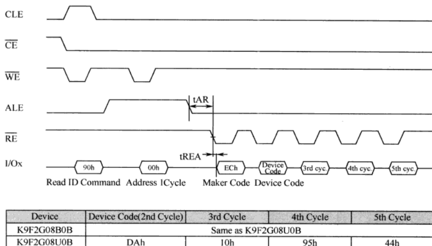
通过本实验，初学者需要对NAND FLASH的基本操作有个大概的了解，明确页写入、页读取的基本实现方法，深入理解块擦除的概念。

12.2.4　NAND FLASH基础实验之读ID

一般对于某一种类的NAND FLASH，在其内部都会保存自身相关的一些信息，如制造商码、产品码以及产品内部信息（如页大小）等。在一般系统设计过程中，就是根据这些信息来设计驱动程序。例如，先读取产品码，根据不同的产品码调用不同的驱动程序，也

就是所谓的支持多种 NAND FLASH 启动。

这些信息一般在数据手册上会有相关的数据，笔者采用的 NAND FLASH 型号为 K9F2G08U0B，结合数据手册中给出的信息，产品码示意图如图 12-17 所示。



Device	Device Code(2nd Cycle)	3rd Cycle	4th Cycle	5th Cycle
K9F2G08B0B	Same as K9F2G08U0B			
K9F2G08U0B	DAh	10h	95h	44h

图 12-17　产品码示意图

可见，5 个产品码分别是 0xEC、0xDA、0x10、0x95、0x44。其中，0xEC 是制造商码，0xDA 是产品码。不同型号的 NAND FLASH，其产品码不同，一般是通过产品码来区分不同的 NAND FLASH。

从图 12-17 中可以得到读取产品码的基本流程：

（1）发送读 ID 命令。

（2）发送地址 0。

（3）延时一会儿。

（4）读取 5 个字节的 ID 即可。

在上述实验的基础上，修改 nand.c 文件，添加读 ID 函数，内容如下：

```
void NF_ReadID(unsigned char *buf)
{
    int     i;

1   NF_Enable();
2   NF_Enable_RB();

3   NF_Send_Cmd(CMD_READID);
4   NF_Send_Addr(0x0);

5   for ( i = 0; i < 100; i++ );

6   *buf        = NF_Read_Byte();
```

```
7    *(buf+1)        = NF_Read_Byte();
8    *(buf+2)        = NF_Read_Byte();
9    *(buf+3)        = NF_Read_Byte();
10   *(buf+4)        = NF_Read_Byte();

11   NF_Disable();
}
```

第 1～2 行，使能 NAND FLASH。

第 3 行，发送读 ID 命令。

第 4 行，发送地址 0。

第 5 行，延时一会儿。为什么需要延时呢，请读者注意图 12-17 中有个时间 tAR。

第 6～10 行，读取 5 个字节的 ID 即可。

Mani.c 文件内容如下：

```
#include "config.h"
#include "uart.h"
#include "nand.h"

unsigned char table[]={0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,
                       0x38,0x39,0x41,0x42,0x43,0x44,0x45,0x46} ;
void IO_Init() ;
int Main()
{
    unsigned char pbuf[5] ;
    unsigned int i ;
    IO_Init() ;
    NF_ReadID(pbuf) ;
    while(1)
    {
        for(i = 0 ; i < 5 ; i++)
        {
            putc(table[pbuf[i]/16]) ;
            putc(table[pbuf[i]%16]) ;
            putc(' ') ;
        }
    }
    return 1 ;
}
void IO_Init()
{
    Uart0_Init(115200) ;
    NF_Init() ;
}
```

初始化之后，调用读产品码函数 NF_ReadID()，然后显示即可。

12.2.5 读 ID 实例测试

将上述工程编译，生成.bin 格式的二进制文件，将其下载到 NAND FLASH 中，打开超级终端，波特率设为 115200，超级终端上显示了程序执行的结果，如图 12-18 所示。

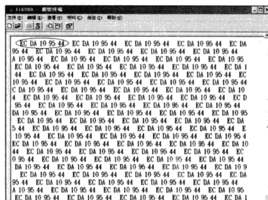


图 12-18 读 ID 实验结果

从图 12-18 中可以看到读取到的产品信息码字。第 1 位是制造商码，第 2 位是产品码，其他 3 位是关于产品的细节描述。下面以第 4 位 0x95 为例，0x95 以十六进制数表示，对应的二进制数是 0b10010101。可见，最低 2 位是 01，这 2 位的含义可以对照数据手册，如图 12-19 所示，01 表示该 NAND FLASAH 页面大小为 2 KB，这与前面的讲述一致。对于其他位的含义，读者可以自己查阅数据手册。

4th ID Date

	Description	I/O7	I/O6	I/O5	I/O4	I/O3	I/O2	I/O1	I/O0
Page Size (w/o redundant area)	1 KB							0	0
	2 KB							0	1
	4 KB							1	0
	8 KB							1	1
Block Size (w/o redundant area)	64 KB			0	0				
	128 KB			0	1				
	256 KB			1	0				
	512 KB			1	1				
Redundant Area Size (byte/512 byte)	8						0		
	16						1		
Organization	x8			0					
	x16			1					
Serial Access Minimum	50 ns/30 ns	0				0			
	25 ns	1				0			
	Reserved	0				1			
	Reserved	1				1			

图 12-19 第 4 位的含义

12.3 NAND FLASH 高级实验

前面对 NAND FLASH 的读、写、擦除进行了讲解。也有很多人说 NAND FLASH 不能以字节为单位进行读、写，但是，这种说法确实是不对的。比如笔者采用的 NAND FLASH 的型号是 K9F2G08U0B 就支持随机读写，也就是支持单字节读写。

随机写操作流程图如图 12-20 所示。

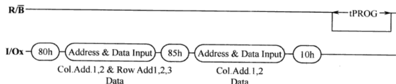


图 12-20　随机写操作流程图

随机写操作基本流程：

（1）发送页写入命令 0x80。

（2）发送页地址。

（3）发送随机写发起命令 0x85。

（4）发送页内地址。

（5）发送随机写确认命令 0x10，NAND FLASH 收到该命令后会自动将数据写入上述地址单元中。

（6）检测忙信号，等待数据写入完成。

注意：因为页地址指的是该页距离第 0 块第 0 页的绝对地址，所以需要 5 个地址周期。页的绝对地址的计算方法与前文讲的一致，即页的绝对地址=块号×64+页号，第 2 次发送地址时，该地址是所写入的数据在该页中的地址，因为每一页大小是 2K+64 字节，所以，地址线需要 A0～A11，因此需要 2 个地址周期。

随机读操作流程图如图 12-21 所示。

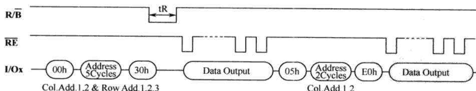


图 12-21　随机读操作流程图

随机读操作基本流程：

（1）发送页读取命令 0x00。

（2）发送页地址。

（3）发送页读取确认命令 0x30，NAND FLASH 会自动将该页数据读取到内部的数据

寄存器中。

（4）发送随机读发起命令 0x05。

（5）发送页内地址。

（6）发送随机读确认命令 0xE0。

（7）检测忙信号，等待数据写入完成。

12.3.1 随机读、写实验代码详解

结合前面的讲解对随机读、写操作的基本流程，编写相应的函数。

在前面实验的基础上，需要做如下改动。

- 在 nand.h 文件中增加随机读、写操作命令的宏定义

```
#define CMD_RANDOMREAD1     0x05    //随机读命令周期 1
#define CMD_RANDOMREAD2     0xE0    //随机读命令周期 2
#define CMD_RANDOMWRITE     0x85    //随机写命令
```

同时增加对随机读、写函数的声明，如下：

```
extern unsigned char NF_RamdomRead(unsigned int block, unsigned int page, unsigned int add) ;
extern unsigned char NF_RamdomWrite(unsigned int block,unsigned int page, unsigned int add, unsigned char dat) ;
```

- 在 nand.c 文件中增加上述两个函数
  - ◆ 随机写操作可用如下函数实现。

```
unsigned char
NF_RamdomWrite(unsigned int block,unsigned int page, unsigned int add, unsigned char dat)
{
1    unsigned int page_number = (block<<6)+page;
2    NF_Enable();
3    NF_Enable_RB();

4    NF_Send_Cmd(CMD_WRITE1);
5    NF_Send_Addr(0x00);
6    NF_Send_Addr(0x00);
7    NF_Send_Addr((page_number) & 0xff);
8    NF_Send_Addr((page_number >> 8) & 0xff);
9    NF_Send_Addr((page_number >> 16) & 0xff);

10   NF_Send_Cmd(CMD_RANDOMWRITE);

11   NF_Send_Addr((char)(add&0xff));
12   NF_Send_Addr((char)((add>>8)&0x0f));
```

```
13  NF_Send_Data(dat);

14  NF_Send_Cmd(CMD_WRITE2);

15  NF_Check_Busy();
16  NF_Disable();

}
```

第 1 行，计算页的绝对地址。

第 2～3 行，打开 NAND FLASH，同时开启忙信号检测功能，以后就可以对 NAND FLASH 进行操作，然后通过检测忙信号来获取 NAND FLASH 内部的工作状态。

第 4 行，发送页写入发起命令 0x80。

第 5～9 行，发送页绝对地址。

第 10 行，发送随机写命令 0x85。

第 11～12 行，发送页内地址。

第 13 行，发送要写入的数据。

第 14 行，发送页写入确认命令 0x10，NAND FLASH 收到该命令后，会自动将数据写入相应的地址处。

第 15 行，检测忙信号，等待操作完成。

第 16 行，操作完成后，关闭片选信号即可。

◆ 随机读操作可用如下函数实现。

```
unsigned char NF_RamdomRead(unsigned int block, unsigned int page, unsigned int add)
{
    unsigned char buf ;
1   unsigned int page_number = (block<<6) + page;

2   NF_Reset();
3   NF_Enable();
4   NF_Enable_RB();

5   NF_Send_Cmd(CMD_READ1);

6   NF_Send_Addr(0x00);
7   NF_Send_Addr(0x00);
8   NF_Send_Addr((page_number) & 0xff);
9   NF_Send_Addr((page_number >> 8) & 0xff);
10  NF_Send_Addr((page_number >> 16) & 0xff);

11  NF_Send_Cmd(CMD_READ2); //页读命令周期 2----0x30
```

```
12  NF_Check_Busy();

13  NF_Send_Cmd(CMD_RANDOMREAD1);

14  NF_Send_Addr((char)(add&0xff));
15  NF_Send_Addr((char)((add>>8)&0x0f));

16  NF_Send_Cmd(CMD_RANDOMREAD2);

17  NF_Check_Busy();

18  buf = NF_Read_Byte() ;

19  NF_Disable();

20  return buf;
}
```

第 1 行，计算页的绝对地址。

第 2～4 行，打开 NAND FLASH，同时开启忙信号检测功能，以后就可以对 NAND FLASH 进行操作，然后通过检测忙信号来获取 NAND FLASH 内部的工作状态。

第 5 行，发送页读取发起命令 0x00。

第 6～10 行，发送页绝对地址。

第 11 行，发送页读取确认命令 0x30。

第 12 行，检测忙信号，等待 NAND FLASH 内部操作完成。

第 13 行，发送随机读发起命令 0x05。

第 14～15 行，发送页内地址。

第 16 行，发送随机读确认命令 0xE0。

第 17 行，检测忙信号，等待操作完成。

第 18 行，从 S3C2440 NAND FLASH 特殊功能寄存器中读取数据即可。

第 19 行，关闭片选信号。

第 20 行，将读取的数据返回。

Main.c 文件内容如下：

```
#include "config.h"
#include "uart.h"
#include "nand.h"

unsigned char table[]={0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,
                       0x38,0x39,0x41,0x42,0x43,0x44,0x45,0x46} ;
```

```
void IO_Init() ;

int Main()
{
      unsigned char recv ;

      IO_Init() ;

1     NF_EraseBlock(17) ;
2     NF_RamdomWrite(17,6,6,0xA0) ;
3     recv = NF_RamdomRead(17,6,6) ;

      while(1)
      {
4           putc(table[recv/16]) ;
5           putc(table[recv%16]) ;
6           putc('  ') ;

      }
      return 1 ;
}
void IO_Init()
{
      Uart0_Init(115200) ;
      NF_Init() ;
}
```

该文件内容与前面实验部分内容基本相同，对基本的初始化部分不再赘述。

第 1 行，擦除第 17 块，每次写操作之前必须进行擦除操作。

第 2 行，调用随机写函数 NF_RamdomWrite()向第 17 块中第 6 页 6 个存储单元中写入数据 0xA0。

第 3 行，调用随机读函数 NF_RamdomRead()将第 17 块中第 6 页 6 个存储单元中的数据读出。

第 4～6 行，将读出的数据打印到串口。

12.3.2 随机读、写实例测试

将上述工程编译，生成.bin 格式的二进制文件，将其下载到 NAND FLASH 中，打开超级终端，波特率设为 115200，超级终端上显示了程序执行的结果，如图 12-22 所示。

可见，读出的数据确实是 0xA0，到此为止，本实验结束。

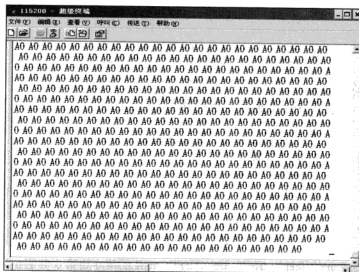


图 12-22　随机读、写实验

12.4　回顾启动代码中的 NAND FLASH 读取函数

在第 7 章讲解启动代码时讲解到，从 NAND FLASH 启动时需要将 NAND FLASH 中的代码复制到内存中，内存的起始地址为 0x30000000。这里使用的函数是 RdNF2SDRAM()。这个函数又是怎么实现的呢？

该函数代码如下：

```
     void RdNF2SDRAM(void)
     {
          unsigned int i;
1    unsigned int start_addr = 0x0;
2    unsigned char * dstaddr = (unsigned char *)0x30000000;
3    unsigned int size = 0x100000;

4    NF_Init();

5    switch(rNF_ReadID())
          {
6         case 0xD6:
7              for(i = (start_addr >> 9); size > 0; )
               {
8                   SB_ReadPage(i, dstaddr);
9                   size -= 512;
10                  dstaddr += 512;
```

```
11                  i ++;
              }
12            break;

13        case 0xDA:
14            for(i = (start_addr >> 11); size > 0;i ++ )
              {
15                NF_ReadPage(i/64,i%64, dstaddr);
16                size -= 2048;
17                dstaddr += 2048;
              }
18            break;
        }
    }
```

第1行，定义起始地址为0，即从NAND FLASH的第0块第0页第0个存储单元开始复制。

第2行，定义了一个指向内存首地址的指针。

第3行，定义了需要复制的数据量大小，这里默认是4 MB，如果用户程序超过4 MB，则需要修改这里。

第4行，初始化NAND FLASH。

第5行，调用读取产品码函数，然后使用switch语句，根据不同类型的NAND FLASH选择不同的读取函数。这里读取产品码的函数是rNF_ReadID()，该函数与前面讲的NF_ReadID基本一致，只不过这里只需要读取NAND FLASH的产品码即可，对于其他4个参数不关心，所以该函数最后只是返回了NAND FLASH的产品码。每种NANF FLASH的产品码是不同的，所以，使用该产品码就可以区分不同的NAND FLASH。

rNF_ReadID()函数如下，与前文所讲的NF_ReadID基本一致，在此不再赘述。

```
char rNF_ReadID()
{
    char  pMID;
    char  pDID;
    char  nBuff;
    char  n4thcycle;
    char  n5thcycle;
    int   i;
    NF_Enable();
    NF_Enable_RB();
    NF_Send_Cmd(CMD_READID);
    NF_Send_Addr(0x0);

    for ( i = 0; i < 100; i++ );
```

```
        pMID        = NF_Read_Byte();
        pDID        = NF_Read_Byte();
        nBuff       = NF_Read_Byte();
        n4thcycle   = NF_Read_Byte();

        NF_Disable();
        return (pDID);
    }
```

第 6～12 行，这几行不会执行，因为笔者采用的 NAND FLASH 型号为 K9F2G08U0B，其产品码为 0xDA。

第 13～18 行，这才是该函数的关键。因为 NF_ReadPage()函数是以页为单位进行读取的，每次读取 2KB 数据，如图 12-23 所示，因此，数据指针的移动是页对齐的，即每次移动一页。

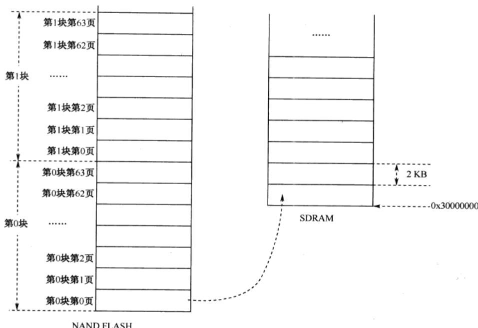


图 12-23　代码复制图解

第 14 行，start_addr >> 11 位，相当于 start_addr/2048（前文讲过应尽量使用移位运算代替除法运算，主要是为了节约运算时间），这样就得到了给定地址所对应的页。

第 15 行，这里需要注意，i 指的是绝对地址，因为每块包含 64 页，i/64 得到给定地址处于哪一块；i%64 是该页在块内的第几页。

注意：这里涉及一个绝对地址和相对地址的概念。比如，第 66 页，如果是从第 0 块第 0 页开始计数，则是第 66 页；如果从第 1 块第 0 页开始计数，则是第 2 页，如图 12-24 所

示。函数 NF_ReadPage(unsigned int block,unsigned int page, unsigned char * dstaddr)的参数就是需要知道该页处于第几块，然后是该块中的第几页，因此采用了上面的转换关系。

第 16 行，因为在第 15 行中调用页读取函数已经复制了 2 KB 的数据，所以要将计数器调整，size 记录的是还有多少数据没有复制完。

第 17 行，将内存地址指针移动 2 KB，因为已经复制了 2 KB 的数据，所以要将指针调整。然后就是循环执行上述操作，直到将 4MB 数据全部复制完毕。

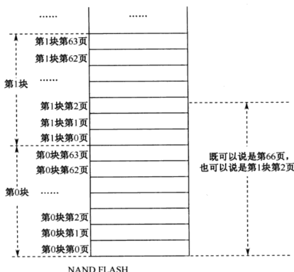


图 12-24　页的绝对地址与相对地址

12.5　本章小结

本章主要对 NAND FLASH 进行了讲解，主要是向初学者展示如何对 NAND FLASH 进行基本的读、写、擦除操作，从原理入手，结合具体的数据手册，带领读者一步一步地完成 NNAD FLASH 驱动函数的编写。此外，结合具体的实验，验证驱动函数的正确性。

当然，在具体的应用中，NAND FLASH 一般需要特定文件系统的支持，这样使用起来比较方便，但是无论什么文件系统都离不开底层驱动函数的学习，读者在后续的学习过程中会接触到文件系统，到时可以系统地学习。

第13章 LCD控制器原理与实验

数据的存储与显示是嵌入式系统的常见功能。在上一章中，对 NAND FLASH 的原理进行了讲解。本章将对 LCD 进行讲解，从基本原理入手，最终讲解汉字编码与显示的基本原理。

S3C2440 LCD 控制器的特性如下。

对于 STN 屏，特性如下。

- 支持 3 种扫描方式：4 位单扫、4 位双扫和 8 位单扫。
- 支持单色、4 级灰度和 16 级灰度屏。
- 支持 256 色和 4 096 色彩色 STN 屏（CSTN）。
- 支持分辨率为 640 像素×480 像素、320 像素×240 像素、160 像素×160 像素显示模式。

对于 TFT 屏，特性如下：

- 支持单色、4 级灰度、256 色的调色板显示模式。
- 支持 64K 和 16M 色非调色板显示模式。
- 支持分辨率为 640 像素×480 像素、320 像素×240 像素、160 像素×160 像素显示模式。

13.1 LCD 和 LCD 控制器工作原理

物质分为三种状态：固态、液态和气态。但是，有些物质的液态还可以细分，其中分子排列具有方向性的液体称为“液态晶体”，简称“液晶”。

固态晶体通常具有方向性。液态晶体不仅具有一般晶体的方向性，还具有液体的流动性。液晶的方向性可由外加的电场或磁场来控制，当光线入射到液晶中时，光线会按照液晶分子排列方向发生偏转现象。

一般电子产品中所用液晶显示器，就是利用液晶光电效应，即由外部电压控制，利用透过液晶分子折射特性，以及对光线旋转能力来获得亮暗变化的情况，进而达到显像的目的。

STN 和 TFT 都是使用一种称为“向列型（Nematic）”液晶的物质，它呈丝状，利用电场来控制“丝状”液晶的方向，进而达到显像的目的。

13.1.1 LCD 概述

利用液晶制成显示器称为液晶显示器，即 LCD（Liquid Crystal Display）。按照驱动方式的不同，可以将液晶显示器分为静态驱动（Static）、单纯矩阵驱动（Simple Matrix）

和主动矩阵驱动（Active Matrix）3 种。其中，单纯矩阵型又被称为被动式（Passive），包括扭转向列型（Twisted Nematic，TN）和超扭转式向列型（Super Twisted Nematic，STN）两种。较为常见的主动矩阵型是薄膜式晶体管型（Thin Film Transistor，TFT），即所谓的 TFT 液晶。

结合 TN 和 STN 型的液晶的显示原理可知，当显示部分做得很大时，中心部分的电极反应时间会变长。因为目前的手机显示屏都比较小，液晶反应时间的影响对于手机来讲并不是很大的问题。但是，对于笔记本等需要大屏幕液晶显示器的设备来说，液晶反应时间太慢将严重影响显示效果，因此，TFT 液晶技术引起了厂商的注意。

TFT 液晶为每个像素都设有一个半导体开关，其加工工艺类似于大规模集成电路。一般通过点脉冲直接控制每个像素，所以，每个像素点是相对独立的，这种设计极大地提高了显示屏的反应速度，同时可以精确控制显示灰度，所以 TFT 液晶的色彩更鲜艳。

笔者开发板上使用的液晶型号是 WXCAT35-TG3（在液晶显示器的背面可以看到该型号），即东华 3.5 英寸 TFT 真彩液晶。下面讲述的内容全是以该液晶为主，由于 TFT 液晶的驱动原理都相似，因此，如果读者开发板上是其他类型的液晶，也可以此作为参考。

如果读者是初次接触 TFT LCD，那么笔者建议按照如下思路来学习。

（1）了解 TFT LCD 的接口信号，了解在 TFT LCD 上显示一副图像的原理。

（2）熟悉 S3C2440 处理器提供的 LCD 控制器中需要初始化哪些寄存器。

（3）在 TFT LCD 上显示一个像素，然后显示一副单色图像。

本章也是按照上述思路来讲解，至于显示模式、显示彩色图片等问题是后续的事情，尤其是初学阶段，重要的是掌握初始化和显示的基本流程。切记不要将过多的精力放在不同的显示模式上，毕竟在应用中受到很多限制，要根据实际情况选择显示模式。但是，只要掌握了一种显示模式的操作，其他显示模式原理类似。

13.1.2 LCD 接口信号

对 TFT LCD 的访问需要用到的信号，如表 13-1 所示，请读者注意，TFT LCD 接口信号是一样的，只要按照一款 LCD 屏参数正确初始化后，很容易掌握驱动其他类型的 LCD 的方法。

表 13-1　TFT LCD 接口信号

信　号	功 能 描 述
VSYNC	垂直同步信号
HSYNC	水平同步信号
VCLK	像素时钟
LEND	行结束信号
VD[0:23]	数据输入信号

每个信号的含义如下：

- VSYNC 信号表示显示新一帧图像数据的开始。
- HSYNC 信号表示显示一行的开始。

- VCLK 信号表示像素时钟，每个 VCLK 周期显示一个像素。
- LEND 信号表示一行的结束。
- VD[0:23]信号表示像素数据输入。

13.1.3 LCD 显示原理

图像的显示方式分为单色显示和彩色显示。

单色显示时，只有黑白两种颜色，因此使用一个二进制位表示即可，0 表示黑颜色，1 表示白颜色，也就是所谓的 1 BPP（Bit Per Pixel，像素/位）。

彩色显示模式又分为不同的颜色模式，常见的是 256 色显示模式，每个像素使用 8 个二进制位表示不同的颜色等级，即所谓的 8 BPP。

一般图像的显示模式如表 13-2 所示，图像深度即表示一个像素所需要的二进制位数。一般图像深度越大，能表示的颜色数目越多，显示的图像色彩越丰富。但是，存储该图像所需的存储器空间也越大。由于人眼对颜色的分辨能力有限，一般情况下不需要特别大的图像深度，可以通过适当降低图像深度来节省图像的存储器空间。

表 13-2 一般图像的显示模式

图像深度	颜色数量	名称
1	2	黑白图像
4	16	16 色图像
8	256	256 色图像
16	65 536	增强色图像
24	16 777 216	真彩色图像

在 LCD 上，完整的一幅图像称为一帧（Frame），每帧图像由很多行组成，每行由很多个像素组成，每个像素使用很多位二进制数来表示，如 8 BPP。

在 LCD 上显示图像的原理和 CRT 显示器的显示原理相同，都是采用“之”字形方式扫描，如图 13-1 所示。

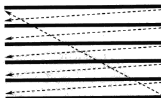


图 13-1 “之”字形扫描

“之”字形扫描原理：从 LCD 的左上角开始，一行一行地取出每个像素的数据并将其显示在屏幕上，如图 13-1 中粗实线所示。当显示到一行的最右边一个像素时，自动跳到下一行的最左边显示，如图 13-1 中虚线所示。当显示到最后一行时，跳到屏幕的左上角，开始显示下一帧图像数据，如图 13-1 中从右下角指向左上角的虚线所示。

在扫描过程中需要使用表 13-1 中的 HSYNC、VSYNC 信号控制扫描路线，HSYNC 表示显示一行的开始，即跳到最左边开始新一行的扫描；VSYNC 表示显示新一帧图像数据的开始，即跳到屏幕的左上角开始新一帧图像数据的扫描。

此外，在显示器显示图像时，会在显示器的边缘部分看到黑色的边框，如图 13-2 所示，图中四周存在黑色的边框，具体原因如下：

- 上方的边框是因为发出 VSYNC 信号后，扫描几行后，才会显示有效数据。
- 下方的边框是因为显示完一帧有效数据后，VSYNC 还没有发出，所以继续扫描而

产生的。

- 左方的边框是因为发出 HSYNC 信号后，扫描几列后，才显示第一列有效数据。
- 右方的边框是因为显示完一行有效数据后，HSYNC 还没有发出，所以继续扫描而产生的。

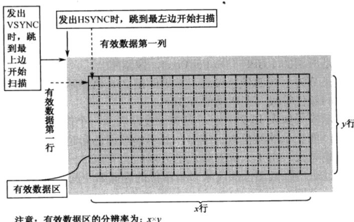


图 13-2　显示器显示原理图解

通过前面的讲解，不难发现，VSYNC 信号出现表示显示一帧图像，那么 VSYNC 的频率即 1s 内能显示多少帧图像，所以这就是所谓的显示器频率，也叫做场频率或者垂直频率；HSYNC 的频率即水平频率。

显示分辨率又称屏幕分辨率，是指显示器屏幕上显示的有效像素点的数目，如图 13-2 所示。

众所周知，光的三原色是红、绿、蓝，将这三种颜色按照不同的比例混合就可以得到众多的颜色，因此，才称这三种颜色为原色。例如，在一个黑屋子里有红色、绿色和蓝色三种颜色的灯，当三种灯都不打开时，屋子里是黑颜色。打开红灯时，屋子里是红颜色，接着打开蓝颜色的灯，屋子里显示红色与蓝色混合后的颜色。最后在打开绿色的灯时，在三种颜色相交的地方是白色，即白颜色是三原色按照 1∶1∶1 的比例混合得到的，其他颜色的形成过程与此类似。

在 LCD 上显示一个像素数据就像三原色的混合。首先确定表示每种颜色所需要的二进制位数。例如，使用 4 位二进制数表示红色，可以将红色分为 16 个等级，从第 0 级红色到第 16 级红色，红色逐渐加深，其他三种原色也是分别使用 4 位二进制数表示，则总共有 16 级红色、16 级绿色和 16 级蓝色，这三种不同级别的颜色可以自由叠加，共有 16×16×16=4 096 种组合，也即总共可以表示 4 096 种颜色。

LCD 可以支持单色（1 BPP）、4 级灰度（2 BPP）、16 级灰度（4 BPP）和 256（8 BPP）调色板显示模式，同时也支持 16 BPP 和 24 BPP 的非调色板显示模式。下面以 16 BPP 显示模式进行讲解。所谓的 16 BPP 就是使用 16 个二进制位表示一个像素的颜色。前文讲到，每个像素是三种原色的混合，那么这 16 位数据如何表示这三种原色呢？有两种表示模式，如图 13-3 所示。

（5:6:5）

VD	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RED	4	3	2	1	0		NC								NC								NC	
GREEN									4	3	2	1	0	I										
BLUE															4	3	2	1	0	I				

（a）5:6:5 显示模式

（5:6:5）

VD	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RED	4	3	2	1	0	I	NC								NC								NC	
GREEN									4	3	2	1	0	I										
BLUE															4	3	2	1	0	I				

（b）5:5:5:1 显示模式

图 13-3　16 BPP 显示模式

- 5:6:5 显示模式

该模式使用高 5 位 VD[23:19]表示红色，中间 6 位 VD[15:10]表示绿色，VD[7:3]表示蓝色。

- 5:5:5:1 显示模式

该模式使用高 6 位 VD[23:18]表示红色，其中前 5 位表示红色的级别，第 6 位表示红色的透明度；VD[15:10]表示绿色，其中前 5 位表示绿色的级别，第 6 位表示绿色的透明度；VD[7:2]表示蓝色，其中前 5 位表示蓝色的级别，第 6 位表示蓝色的透明度。

注意：因为 LCD 的数据信号输出是 VD[0:23]共 24 位，但是对于不同的显示模式可能只使用了这 24 位中的某些位，如图 13-3 中的 NC 表示该显示模式下，该位没有使用。

本书讲解过程中使用的显示模式是 16 BPP，5:6:5 显示模式。

13.1.4　LCD 操作时序详解

下面结合时序图，讲解 TFT LCD 的显示原理，不同类型的 TFT LCD，其显示原理类似。

概括地讲，TFT LCD 显示图像的原理是：采用“之”字形扫描路线，每个像素时钟 VCLK 显示一个像素的数据，使用 HSYNC 信号控制切换到下一行从最左边开始扫描，使用 VSYNC 信号控制切换到左上角开始扫描显示一帧新的图像。

显示一帧图像数据的时序图如图 13-4 所示，具体过程如下：

（1）VSYNC 信号有效时，表示开始显示一帧数据。

（2）VSPW 表示 VSYNC 信号的脉冲宽度为（VSPW+1）个 HSYNC 信号周期，即这（VSPW+1）行数据是无效的。

（3）VSYNC 信号脉冲之后，要经过（VBPD+1）个 HSYNC 信号周期，开始传输有效数据，所以在 VSYNC 信号有效之后，总共还要经过（VSPW+1+VBPD+1）个无效行，第一个有效行才出现，它对应图 13-2 上方的黑色边框。

（4）随后发送 LINEVAL+1 行有效数据。

（5）最后是（VFPD+1）个无效行，它对应图 13-2 下方的黑色边框，完整的一帧结束，紧接着就是下一帧数据的开始（下一个 VSYNC 信号）。

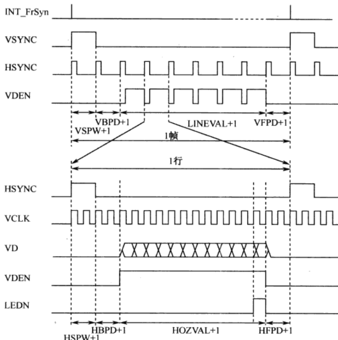


图 13-4 TFT LCD 时序图

显示一帧图像数据是一行一行地显示的，对每一行图像数据而言，具体的显示过程如下：

（1）HSYNC 信号有效表示开始显示一行数据。

（2）HSPW 表示 HSYNC 信号的脉宽为（HSPW+1）个 VCLK 信号周期，即（HSPW+1）个像素，这（HSPW+1）个像素是无效的。

（3）HSYNC 信号脉冲之后，要经过（HBPD+1）个 VCLK 信号周期，有效数据才会出现，所以在 HSYNC 信号有效之后，总共还要经过（HSPW+1+HBPD+1）个无效像素，第一个有效像素才出现，它对应左边的黑色边框。

（4）发送 HOZVAL+1 个像素的有效数据。

（5）（HFPD+1）个无效像素，它对应图 13-2 右边的黑色边框，完整的一行结束，紧接着就是下一行数据的开始（下一个 HSYNC 信号）。

在上面的讲解过程中提到了像素时钟信号 VCLK。对于 S3C2440 处理器来说，VCLK 又是怎么得到的呢？VCLK 信号是通过 HCLK 信号分频得到的，如图 13-5 所示，具体的分频系数可以通过寄存器中的值 CLKVAL 进行改变。

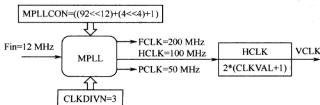


图 13-5 VCLK 的来源

注意：对于 FCLK、HCLK 和 PCLK 的产生及初始化过程已在第 8 章“系统时钟和定时器”中进行了详细的讲解，在此不做赘述。

VSYNC 信号的频率即显示器的帧频率（场频率），计算公式如下：

帧频率 = 1/ [{ (VSPW+1) + (VBPD+1) + (LINEVAL + 1) + (VFPD+1) } * {(HSPW+1) + (HBPD +1)+ (HFPD+1) + (HOZVAL + 1) } * { 2 * (CLKVAL+1) / (HCLK) }]

到此可以说，TFT LCD 的显示原理已基本讲清楚（当然，上述公式中的很多参数还没有确定，但是这只需要按照时序图中给出的参数值计算即可），基本流程总结如下：

（1）初始化上述时序信号的各项参数（下面的讲解主要围绕这个问题展开）。

（2）将显示图像数据写入帧内存，然后将帧内存的地址告诉 TFT LCD，LCD 会自动从帧内存中读取帧内存中的数据将其显示到屏幕上。

13.1.5 S3C2440 LCD 控制器

要使一块 LCD 屏显示图像，需要 LCD 驱动器和 LCD 控制器。

- 通常，LCD 驱动器与 LCD 玻璃基板制作在一起。
- LCD 控制器需要外部电路来实现。

S3C2440 内部集成了 LCD 控制器，可以很方便用于控制各种类型的 LCD 屏，包括 STN 屏和 TFT 屏，这将大大加快产品的开发。

本章是驱动 TFT 屏，对于驱动 TFT 屏，需要初始化 LCD 控制器使其产生正确的时序（类比在前一章中，驱动 NAND FLASH 就是初始化 NAND FLASH 控制器，使其产生正确的时序）。

S3C2440 LCD 控制器的结构如图 13-6 所示，其中 REGBANK 是寄存器组，含有 17 个寄存器以及一块 256×16 的调色板内存。

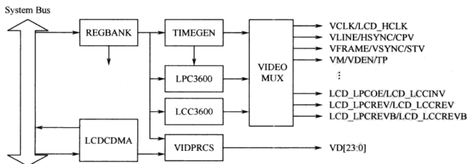


图 13-6 S3C2440 LCD 控制器的结构

在图 13-6 中，REGBANK 包含了初始化 LCD 控制器所需要的所有寄存器，对这些寄存器进行正确的初始化，TIMEGEN 即可按照这些参数产生相应的时序信号，如 VSYNC、HSYNC、VCLK 等；LCDCDMA 是 LCD 专用的 DMA 通道，以 DMA 的方式取得图像数据；VIDPRCS 将图像数据组合成特定的显示格式通过数据总线 VD[0:23]发送

给 LCD。

注意：LPC3600 是为三星公司（Samsung Electronics Company，SEC）的 TFT LCD 设计的。如果不是使用 SEC TFT LCD，可以不用考虑。

S3C2440 LCD 控制器可以应用于多种类型的 LCD，包括 TFT LCD 和 STN LCD，虽然 REGBANK 组中有 17 个寄存器，但是对于 TFT LCD 所使用的寄存器只有 9 个，如表 13-3 所示。

表 13-3　LCD 控制寄存器

寄 存 器	功 能 描 述
LCDCON1～LCDCON5	垂直同步信号
LCDSADDR1～LCDSADDR3	水平同步信号
TPAL	像素时钟

13.1.6　LCD 控制寄存器初始化

下面结合具体的寄存器，讲解各个寄存器的初始化过程。

小技巧：笔者结合自身的经历，并没有对寄存器中的所有位进行展开讲解，使用到哪些位就对哪些位进行了讲解，对没有使用到的位或者初学阶段不需要关心的位没有进行讲解，这样可以最大限度地将注意力集中在学习的重点上，即需要初始化哪些位，而不是刚刚接触该寄存器就面对繁多的控制位。这样做的好处是：当熟悉了部分位的含义后，需要扩展其他功能时，很容易就可以找到相应的控制位。

1．LCDCON1

该寄存器的主要功能是：设置 LCD 的类型、像素时钟 VCLK 和使能 LCD 等，各位的含义如图 13-7 所示。

LCDCON1	位	功 能 描 述
CLKVAL	[17:8]	设置像素时钟，VCLK=HCLK/（2*（CLKVAL+1））
PNRMODE	[6:5]	选择 LCD 的类型，对于 TFT LCD，该位为 0b11
BPPMODE	[4:1]	选择 BPP，对于 TFT LCD 0b1000=1 BPP 0b1001=2 BPP 0b1010=4 BPP 0b1011=8 BPP 0b100=16 BPP 0b1101=24 BPP
ENVID	[0]	LCD 控制信号输出使能位，1：使能　0：禁止

图 13-7　LCDCON1 寄存器

初始化该寄存器主要是确定 CLKVAL 的值，该值主要用于产生 LCD 的像素时钟。查询 LCD 数据手册得到该液晶的典型工作频率是 6.5 MHz，又因为 VCLK=HCLK/(2*(CLKVAL+1))，所以可以取 CLKVAL=7。

可以使用如下代码初始化寄存器 LCDCON1：

rLCDCON1 = (7<<8) | (3<<5) | (12<<1) | (1<<0);

其功能是：产生 LCD 像素时钟，同时选择 16 BPP 显示模式。

2. LCDCON2

主要用于产生垂直方向的时序参数，其各位的含义如图 13-8 所示。

LCDCON2	位	功 能 描 述
VBPD	[31:24]	VSYNC 信号后，经过 VBPD+1 个 HSYNC 信号周期，开始显示有效数据
LINEVAL	[23:14]	LCD *Y* 方向宽度：LINEVAL+1 行
VFPD	[13:6]	一帧显示结束后，需要经过 VFPD+1 行无效数据，才出现 VSYNC 信号
VSPW	[5:0]	VSYNC 信号宽度为 VSPW+1 个 HSYNC 信号周期

图 13-8　LCDCON2 寄存器

一般对于 320 像素×240 像素的分辨率，*Y* 方向即垂直方向，垂直方向显示 240 行像素，所以，LINEVAL+1=240，则可以得到 LINEVAL=239。

3. LCDCON3

主要用于产生水平方向的时序参数，其各位的含义如图 13-9 所示。

LCDCON3	位	功 能 描 述
HBPD	[25:19]	HSYNC 信号后，经过 HBPD+1 个 HSYNC 信号周期，开始显示有效数据
HOZVAL	[18:8]	LCD *X* 方向宽度：HOZVAL+1 行
HFPD	[7:0]	一行数据显示结束后，需要经过 HFPD+1 个无效像素，才出现 HSYNC 信号

图 13-9　LCDCON3 寄存器

一般对于 320 像素×240 像素的分辨率，*X* 方向即水平方向，水平方向显示 320 列像素，所以，LINEVAL+1=320，则可以得到 LINEVAL=319。

4. LCDCON4

该寄存器主要是初始化一个参数 HSPW，如图 13-10 所示。

LCDCON4	位	功 能 描 述
HSPW	[7:0]	HSYNC 信号脉冲宽度为 HSPW+1 个 VCLK 信号周期

图 13-10　LCDCON4 寄存器

上述三个寄存器 LCDCON2～LCDCON4 主要是用于初始化访问 LCD 的一些时序参数，初始化 LCD 控制器主要就是为了对这几个参数进行初始化。下面结合 LCD 数据手册给出的时序图讲解这些参数的初始化。

WXCAT35-TG3 数据手册给出的时序图如图 13-11 所示。

请读者仔细对比图 13-4 和图 13-11，虽然信号名称稍微有点区别，但是很容易将上述控制信号对应起来。

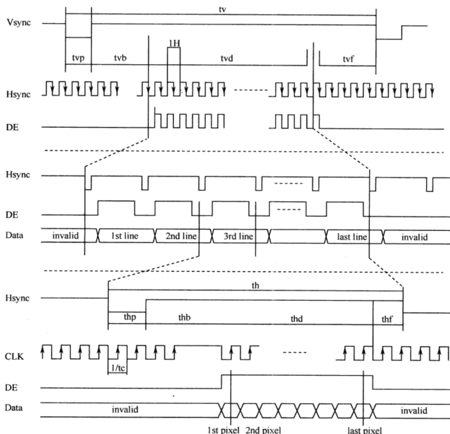


图 13-11　WXCAT35-TG3 数据手册给出的时序图

此外，该数据手册还给出了各个时序参数的具体数值，如图 13-12 所示。

Signal	Item	Symbol	Min	Typ	Max	Unit
Dclk	Frequency	Dclk	-	6.4	-	MHz
	Dclk-Period	Tosc	-	156	-	ns
Data	Setup Time	TSU	12	-	-	ns
	Hold Time	THD	12	-	-	ns
Hsync	Period	th	-	408	-	DCLK
	Pulse Width	thp	-	30	-	DCLK
	Back-Porch	thb	-	38	-	DCLK
	Display Period	thd	-	320	-	DCLK
	Front-Porch	thf	-	20	-	DCLK
Vsync	Period	tv	-	270	-	TH
	Pulse Width	tvp	-	3	-	TH
	Back-Porch	tvb	-	15	-	TH
	Display Period	tvd	-	240	-	TH
	Front-Porch	tvf	-	12	-	TH

图 13-12　时序参数

从图 13-12 中可以得出，该液晶显示器的工作频率为 6.4 MHz，此外可以确定如下参数。

例：结合图 13-4、图 13-11 和图 13-12 确定 VSPW 的值。

- 不难发现，图 13-4 中的像素时钟信号 VCLK 对应于图 13-11 的信号 CLK（对应于图 13-12 中的 Dclk），CLK=6.4 MHz，所以初始化时需要将 VCLK 初始化为 6.4 MHz，又因为 VCLK=HCLK/(2*(VLKVAL+1))，所以可以取 CLKVAL=7。
- 从图 13-4 中得出的结论是：VSPW 表示 VSYNC 信号的宽度是（VSPW+1）个 HSYNC 信号周期。
- 对比图 13-4、图 13-11 不难发现，图 13-4 中的 VSYNC、HSYNC 信号分别对应图 13-11 中的 Vsync、Hsync 信号；从图 13-11 中可以读出结论：Vsync 信号的宽度是 tvp 个 Hsync 信号周期（注意，后面的单位是 TH，1 TH 就是一个 Hsync 信号周期）。
- 从图 13-12 可看到，Tvp=3，即图 13-11 中 Vsync 信号的宽度是 3 个 Hsync 信号周期。
- 回到图 13-4，可以得到 VSPW+1=3，所以 VSPW=2。

注意：这种分析方法是从理论上进行分析，具体的硬件设备可能略有不同。但是，通过这种方法可以将各个参数的值大概确定，有时需要对上述参数进行细微的调整。如果读者比较熟悉上面的方法，可以跳过上述步骤，直接使用下面的公式即可（这里需要注意，尽管可以通过上述分析方法确定各个参数的值，但是在实际使用中需要进行适当的调整）：

- tvp=VSPW+1，得 VSPW=2。
- tvb=VBPD+1，得 VBPD=14。
- tvf=VFPD+1，得 VFPD=11。
- thp=HSPW+1，得 HSPW=29。
- thb=HBPD+1，得 HBPD=37。
- thf=HFPD+1，得 HFPD=19。

5. LCDCON5

前面几个寄存器主要是针对时序信号的时间参数进行的初始化，但是具体信号的极性并没有初始化，而寄存器 LCDCON5 就是针对信号的极性进行初始化。寄存器 LCDCON5 的各位含义如图 13-13 所示。

LCDCON5	位	功能描述
FRM565	[11]	设置 TFT LCD 16 BPP 显示模式　0 表示 5:5:5:5:1 模式　1 表示 5:6:5 模式
INVVCLK	[10]	设置 VCLK 信号的有效沿极性 0：VCLK 下降沿读取数据 1：VCLK 上升沿读取数据
INVLINE	[9]	设置 HSYNC 信号的极性 0:正常极性　　1：翻转极性
INVVFRAME	[8]	设置 VSYNC 信号的极性 0:正常极性　　1：翻转极性
PWREN	[3]	LCD_PWREN 引脚输出使能　0：禁止　1：允许
BSWP	[1]	字节交换使能 0：禁止　　1：使能
HWSWP	[0]	半字交换使能 0：禁止　　1：使能

图 13-13　寄存器 LCDCON5 的各位含义

第 11 位是控制 LCD 的显示模式，笔者采用的是 5:6:5 显示模式。



图 13-14　正极性信号和负极性信号

第 10 位是确定在 VCLK 的上升沿还是下降沿取得像素数据，从图 13-11 中可以看到，在 CLK 信号的上升沿取得像素数据，所以该位应初始化为 1。

什么是信号极性呢？正极性信号和负极性信号如图 13-14 所示。

从图 13-11 中可以看到，Vsync、Hsync 信号都是负极性信号，因此初始化时，第 8、9 两位都应初始化为 1。

第 3 位是控制 LCD 电源引脚是否有效，在基础实验部分进行讲解。

下面还剩下第 0、1 两位，即 HWSWP 和 BSWP，其中 HWSWP 即 Half Word Swap，BSWP 即 Byte Swap。要想初始化这两位还得从帧内存讲起。

什么是帧内存呢？通俗地说，就是一帧图像在内存中的存储地址。本章后面实验部分采用的分辨率是 320 像素×240 像素，即 240 行、320 列像素数据。因此，首先在内存中分配一个二维数组 Buf[240][320]，如图 13-15 所示，显示图像时，数组中的每个元素对应到 LCD 屏幕上的一个像素，则显示该图像就是将二维数组中的元素依次取出，然后显示到 LCD 屏幕对应的位置处。

注意：显示图像时，只需要将帧内存的起始地址告诉 S3C2440 LCD 控制器即可，LCD 控制器会自动使用专用的 DMA 通道读取数据并显示。到此很容易理解，所谓的帧内存就是该数组的首地址。

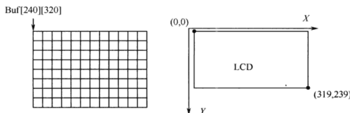


图 13-15　帧内存和图像显示的关系

前文讲到，ARM 处理器默认情况下是小端方式存储，即数据的低位存放着内存地址的低地址处。上面这两个参数就是为了选择大端存储方式还是小端存储方式，如图 13-16 所示，因此，初始化时应该将 BSWP 初始化为 0，HWSWP 初始化为 1 即可。

因此，可以使用如下代码完成此寄存器的初始化：

```
rLCDCON5 = (1 << 11) | (1 << 10) | (1 << 9) | (1<< 8) | (1 << 0);
```

经过上面的讲解，基本完成了对 LCD 控制器控制信号的初始化工作，下面只需要将帧内存的起始地址写入帧内存地址寄存器即可。下面讲解帧内存地址寄存器。

在此之前需要说明的是，虽然 3.5 英寸 LCD 支持的最大分辨率是 320 像素×240 像素，但是可以根据实际情况调整显示范围。实际显示有效图像的区域称为视口（View Port），如图 13-17 所示。

(BSWP=0,HWSWP=0)

	D[31:16]	D[15:0]
000H	P1	P2
004H	P3	P4
008H	P5	P6
…		

(BSWP=0,HWSWP=1)

	D[31:16]	D[15:0]
000H	P2	P1
004H	P4	P3
008H	P6	P5
…		

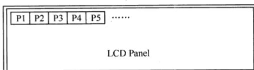


图 13-16 BSWP/HWSWP 的含义

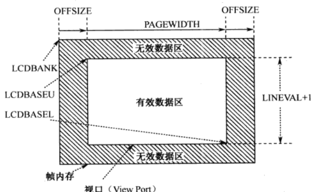


图 13-17 视口（View Port）示意图

小技巧：请读者不要被图 13-17 中这些复杂的显示参数迷惑，这些参数主要是为了更好地适应用户不同显示范围，初学阶段可以使问题简化。比如所有涉及显示范围控制的参数都设为 0，即全屏幕显示，这样学习起来，问题将变得相对容易一些。

考虑到上述最简单的情况下，图 13-17 中的各参数如下：

- PAGEWIDTH=320。
- OFFSIZE= 0。
- LCDBANK = LCDBASEU = LCD_BUFFER。
- LCDBASEL = LCD_BUFFER + 240*320*2（因为每个元素占 2 个字节）。

形象地解释，即 LCDBASEL 就是帧内存的结束地址，也即数组 LCD_BUFFER[240][340]的末地址=数组首地址+元素总个数*每个元素占的字节数

= LCD_BUFFER + 240*320*2。

6. LCDSADDR1

寄存器 LCDSADDR1 主要用于告诉 LCD 控制器帧内存的起始地址，该寄存器的各位含义如图 13-18 所示。

LCDSADDR1	位	功 能 描 述
LCDBANK	[29:21]	保存帧内存的起始地址的高 9 位 A[30:22]
LCDBASEU	[20:0]	对于 TFT LCD，用来存放视口（View Piont）所对应的内存的起始地址 A[21:1]，这块内存也称为 LCD 的帧缓冲区（Frame Buffer）

图 13-18　寄存器 LCDSADDR1

前文提到过，初学阶段不需要使用视口来控制显示范围，全屏显示即可，所以视口地址和帧内存的地址相等。

因此，可以使用如下代码来进行初始化：

```
1    #define LOW21BITS(n)        ((n) & 0x1fffff)
2    volatile unsigned short LCD_BUFFER[240][320];
3    rLCDSADDR1 = ( ( (unsigned int)LCD_BUFFER >> 22) << 21 )   |
                        LOW21BITS((unsigned int)LCD_BUFFER >> 1);
```

第 1 行，定义了一个宏，该宏实现的功能是屏蔽 *n* 的高位，只保留低 21 位。

第 2 行，定义了一个二维数组，数组大小是 240*320，数组的每个元素对应于 LCD 屏幕上的一个像素，这就是所谓的帧内存。这里需要注意，因为是 16 BPP 显示模式，所以每个像素使用 16 个二进制位表示，又因为每个像素对应于该数组的一个元素，所以每个元素占 16 个二进制位，即 2 个字节，所以类型为 unsigned short（对比：unsigned int 占 4 个字节）。

小提示： *为什么使用 volatile 关键字呢？在第 4 章中曾讲解过此关键字的用法。*

第 3 行，数组名代表了该数组的起始地址，所以 LCD_BUFFER 就代表了帧内存的起始地址，这里需要帧内存地址的第 22～30 位（帧内存是 4M 对齐的，所以第 31 位会自动舍弃）写入 LCDSADDR1 的位[29:21]，初学者要注意这里对移位运算的灵活使用。

7. LCDSADDR2

寄存器 LCDSADDR2 主要用于告诉 LCD 控制器视口内存地址的第 1～21 位，如图 13-19 所示，前文讲过，这里的视口地址和帧内存地址是重合的。

LCDSADDR2	位	功 能 描 述
LCDBASEL	[20:0]	对于 TFT LCD，用来存放视口（View Piont）所对应的内存的结束始地址 A[21:1] LCDBASEL=LCDBASEU+(PAGEWIDTH+OFFSIZE)*(LINEVAL+1)

图 13-19　寄存器 LCDSADDR2

因此，可以使用如下代码来进行初始化：

```
rLCDSADDR2 = LOW21BITS( ((unsigned int)LCD_BUFFER + (240 * 320 * 2) ) >> 1 );
```

8. LCDSADDR3

寄存器 LCDSADDR3 主要用于设置 OFFSIZE 和 PAGEWIDTH。注意，这里的单位是以半字为单位的，各位的含义如图 13-20 所示。

LCDSADDR3	位	功能描述
OFFSIZE	[21:11]	表示上一行最后一个有效像素和下一行第一个有效像素之间差值的一半，以半字为单位
PAGEWIDTH	[10:0]	视口的宽度，以半字为单位

图 13-20 寄存器 LCDSADDR3

因此，可以使用如下代码来进行初始化：

```
rLCDSADDR3 = (0 << 11) | (320/ 1);
```

注意：LCD_XSIZE_TFT/1 是什么意思呢？这是初学者经常遇到的问题。这个值都是以半字为单位的，一个字占 4 个字节，半字就是 2 个字节。前文提到过这里使用的显示模式是 16 BPP，即每个像素使用 16 个二进制位来表示，16 个二进制位恰好是 2 个字节，所以，LCDPAGEWIDTH =一行中的像素数目 × 每个像素占用的字节数/2

= 320 * 2 / 2

= 320 / 1

如果使用的是 8 BPP 的显示模式，则每个像素使用 8 个二进制位表示，即 1 个字节，此时 LCDPAGEWIDTH=一行中的像素数目 × 每个像素占用的字节数/2

= 320 * 1 / 2

= 320 / 2

13.2 LCD 基础实验

经过前面的讲解，读者对 TFT LCD 的原理及初始化有了初步的认识。下面结合一个具体的实验，讲解如何在 TFT LCD 上显示一个像素，然后讲解利用 TFT LCD 显示一幅图像的原理。

本实验基本功能：在 TFT LCD 上显示一个像素。

首先需要写初始化函数，然后是显示一个像素。现在的问题是：怎么显示一个像素呢？

TFT LCD 显示图像时，只需要将向帧内存中写入相应的数据，然后将帧内存的地址告诉 S3C2440 处理器，在 LCD 控制器会自动将数据从帧内存中取出来，显示在屏幕上。这里的帧内存就是一个大数组，数组中的每一个元素对应屏幕上的一个像素。

13.2.1 硬件电路分析

TFT LCD 接口信号在前文中已经进行了讲解，S3C2440 处理器内部集成了 LCD 控制器，只需要将控制器输出引脚和 TFT LCD 相应的信号线连接即可，如图 13-21 所示。

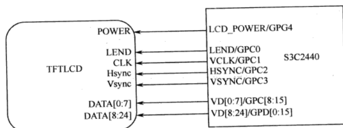


图 13-21　TFT LCD 和 S3C2440 处理器接口电路

从图 13-21 中可以看出，LCD 的电源在 S3C2440 的 GPG4 引脚上。因此，初始化时，需要使该引脚配置为输出，并且输出高电平给 LCD 供电，当不需要显示时，可以使该引脚输出低电平，将 LCD 关闭。

13.2.2　程序代码分析

LCD 基础实验的文件布局如图 13-22 所示。

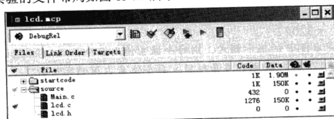


图 13-22　LCD 基础实验的文件布局

LCD 模块包含两个文件：lcd.h 和 lcd.c 文件。

lcd.h 文件内容如下：

```
#ifndef __LCD_H__
#define __LCD_H__

#define LCD_XSIZE_TFT  (320)
#define LCD_YSIZE_TFT  (240)

#define INVVDEN        1     //0=normal        1=inverted
#define HWSWP          1     //Half word swap control
#define PNRMODE        3     //设置为 TFT 屏
#define BPPMODE        12    //设置为 16 BPP 模式

#define CLKVAL_TFT     (7)

#define VBPD           (14)          //垂直同步信号的后肩
#define VFPD           (11)          //垂直同步信号的前肩
#define VSPW           (2)           //垂直同步信号的脉宽
```

```
#define HBPD              (37)          //水平同步信号的后肩
#define HFPD              (19)          //水平同步信号的前肩
#define HSPW              (29)

#define HOZVAL_TFT        (320-1)
#define LINEVAL_TFT       (240-1)       //决定 LCD 显示的行数

#define BPPMODE_TFT                 12

#define FRM565_TFT                  1
#define INVVCLK_TFT                 1
#define INVVLINE_TFT                1
#define INVVFRAME_TFT               1

extern void Lcd_Init(void) ;
extern   void PutPixel(unsigned int x,unsigned int y, unsigned short c ) ;

#endif
```

该文件是对 TFT LCD 初始化参数的一些宏定义，这些参数的含义在前文中已经进行了详细的讲解，最后是声明了两个外部函数：初始化函数 Lcd_Init()和显示一个像素函数 PutPixel（unsigned int x,unsigned int y, unsigned short c）。

小技巧： *有的读者可能会问，为什么需要这些宏定义呢？对此也没有硬性规定，但是当一个器件初始化需要很多参数时，最好使用这种方式，将参数放在一个文件中，这样在调试过程中修改某些参数会比较方便。*

lcd.c 文件是对上述两个函数的具体实现，lcd.c 文件内容如下：

```
#include "2440addr.h"
#include "lcd.h"
1    #define LOW21BITS(n)      ((n) & 0x1fffff)
2    #define Lcd_Enable()      rLCDCON1 |= 1

3    volatile unsigned short LCD_BUFFER[240][320];
```

第 1 行，定义了一个宏，该宏的功能是读取 *n* 的第 0～21 位，将其他位屏蔽。

第 2 行，使用宏定义形式实现对 LCD 控制器的使能。

第 3 行，定义了一个 240×320 的数组，也就是所谓的帧内存，该数组的每一个元素对应于 LCD 屏幕上的一个像素。

```
static void Lcd_Config(void)
{
4    rGPCCON = 0xaaaa02a9;
5    rGPDCON = 0xaaaaaaaa;

6    rLCDCON1 = (CLKVAL_TFT << 8)|(3 <<5 )|(BPPMODE_TFT << 1) ;
```

```
7    rLCDCON2 = (VBPD << 24)|(LINEVAL_TFT << 14)|(VFPD << 6)|(VSPW);
8    rLCDCON3 = (HBPD << 19)|(HOZVAL_TFT << 8)|(HFPD);
9    rLCDCON4 |= (HSPW);
10   rLCDCON5 = (FRM565_TFT << 11) | (INVVCLK_TFT << 10) |
                    (INVVLINE_TFT << 9) | (INVVFRAME_TFT << 8) | (HWSWP);

11    rLCDSADDR1 = (((unsigned int)LCD_BUFFER >> 22) << 21) |
                              LOW21BITS((unsigned int)LCD_BUFFER >> 1);
12    rLCDSADDR2 = LOW21BITS( ((unsigned int)LCD_BUFFER +
                              (LCD_YSIZE_TFT * LCD_XSIZE_TFT *2) ) >> 1 );
13    rLCDSADDR3 = (0 << 11) | (LCD_XSIZE_TFT / 1);
}
```

第4～5行，将GPC和GPD端口配置为LCD信号输出功能。

第6～10行，初始化LCD控制寄存器，各个参数的具体含义在上文中已经进行了详细的讲解。

小技巧：初始化LCDCON1时，请不要使能LCD控制器，即LCDCON1的第0位要置为0，否则LCD显示时会出现“4屏”效果，读者可以实验一下。

第11～13行，初始化帧内存地址寄存器，各个参数的具体含义在上文中已经进行了详细的讲解。

```
static void Lcd_PowerEnable(int powerEnable)
{

14    rGPGCON = rGPGCON & (~(3<<8)) |(3<<8);
15    rGPGDAT = rGPGDAT | (1<<4) ;
16    rLCDCON5 = rLCDCON5 & (~(1<<3)) | (powerEnable<<3);

}
```

该函数实现的功能是：打开LCD电源，因为LCD电源是通过S3C2440处理器的GPG4引脚输出的，所以只需要将GPG4引脚输出高电平即可给LCD供电。

此外，LCDCON5寄存器的第3位控制电源输出是否使能，这主要出于以下原因。

一般可以使用两种方法给LCD供电：第一种是外接电源给LCD供电，在这种情况下，不需要使用GPG4，但是当系统中不需要显示数据时，无法通过软件来关闭LCD的电源；第二种是使用GPG4引脚给LCD供电，这样可以灵活地控制LCD开启与关闭，寄存器LCDCON5的第3位用于配置GPG4引脚给LCD供电使能。

```
void PutPixel(unsigned int x,unsigned int y, unsigned int c )
{
17   if ( (x < 320) && (y < 240) )
18       LCD_BUFFER[(y)][(x)] = c;
}
```

PutPixel(unsigned int x,unsigned int y, unsigned int c)函数的功能是向LCD屏幕上显示一个像素。

第 17 行，判断显示像素的位置是否合适。

第 18 行，如果是在 LCD 屏幕范围内，只需要将数组 LCD_BUFFER 中的对应元素写入相应的数值即可。这样，LCD 控制器会自动读取该数据，然后将其显示在 LCD 屏幕上。

```
void Lcd_Init(void)
{
19      Lcd_Config() ;
20      Lcd_Enable() ;
21      Lcd_PowerEnable(1);
}
```

第 19 行，调用 LCD 端口配置函数 Lcd_Config()，初始化 LCD 所需要的控制信号。

第 20 行，开启 LCD 控制器。注意，在初始化阶段，最好不要开启 LCD 控制器，否则容易出现“4 屏”显示现象。

第 21 行，开启 LCD 电源。到此，LCD 初始化完毕。

Main.c 文件内容如下：

```
#include "lcd.h"
int Main()
{
        Lcd_Init() ; ;
        while(1)
        {

                PutPixel(300,10,569) ;
                PutPixel(40,200,569) ;
        }
        return 0;
}
```

首先调用 LCD 初始化函数 Lcd_Init()对其进行初始化，然后调用显示一个像素函数 PutPixel()函数显示一个像素。

13.2.3 实例测试

将上述工程编译，生成.bin 格式的二进制文件，将其下载到 NAND FLASH 中，启动开发板，发现在 LCD 上显示了 2 个蓝色的点，因此可以推断出：PutPixel(unsigned int x,unsigned int y, unsigned int c)函数的第 3 个参数 569 对应的颜色值是蓝色，当然读者也可以改为其他值，这时 LCD 上显示的就是该值对应的颜色值。

13.3 LCD 基础实验之单像素显示

上文的讲述思路主要是想给初学者展示一种学习思路：学习新的器件时，尽量使问题简单化，慢慢地扩展，最终实现相对较为复杂的功能。上述实验完成了对 LCD 的初始化，然

后在 LCD 上显示了一个像素，下面将其稍微扩展一下，即将 LCD 整个屏幕显示一种颜色。

在 LCD 上显示一个像素，采用的方法是对数组 LCD_BUFFER[240][320]中元素赋值的方法，那么整个屏幕显示一种颜色，只需要将上述数组的所有元素赋为相同的值即可。

13.3.1 程序代码分析

下面讲解 LCD 输出单色图像函数，即使 LCD 屏显示单色。

在 lcd.c 文件中添加如下函数：

```
void Lcd_ClearScr( unsigned int c)
{
    unsigned int x,y ;

    for( y = 0 ; y < 240 ; y++ )
    {
        for( x = 0 ; x < 320 ; x++ )
        {
            LCD_BUFFER[y][x] = c ;
        }
    }
}
```

基本思路是：使用 for 循环对 LCD_BUFFER[240][320]的所有元素赋值。

Main.c 文件内容如下：

```
#include "lcd.h"
int Main()
{
    Lcd_Init() ; ;
    while(1)
    {

        Lcd_ClearScr (569) ;
    }
    return 0;
}
```

首先调用 LCD 初始化函数 Lcd_Init()对其进行初始化，然后调用 LCD 输出单色图像函数 Lcd_ClearScr ()。

13.3.2 实例测试

将上述工程编译，生成.bin 格式的二进制文件，将其下载到 NAND FLASH 中，启动开发板，发现在 LCD 整个屏幕中充满了蓝色，当然读者也可以改为其他值，这时 LCD 上显

示的就是该值对应的颜色值。

13.4 LCD 基础实验之图片显示

到此为止，对 LCD 工作原理进行总结。

基本原理：按照LCD时序信号和显示模式的要求，正确初始化LCDCON1～LCDCON5、LCDSADDR1～LCDSADDR3 寄存器，将帧内存的地址告诉 LCD 控制器，然后只需要将所要显示的数据写入帧内存即可，LCD 控制器会使用专用的 DMA 通道从帧内存取得数据，并将其显示在 LCD 屏幕上。

初始化：

- 配置 GPC、GPD 引脚为 LCD 信号功能。
- 对照 LCD 数据手册，初始化 LCDCON1～LCDCON5 寄存器，特别需要注意时序参数的初始化。
- 按照显示模式，初始化 LCDSADDR1～LCDSADDR3 寄存器。
- 开辟一个帧内存，其实就是定义一个大数组。

显示数据：将要显示的数据写入帧内存即可。在前面基础实验中已经实现了向 LCD 屏幕输出一幅单色的图像，由此可以想到，如果使 LCD 显示一幅图像，只需要将该图像转换为一个和屏幕尺寸相同的数组（此数组中的每个元素都是从源图像中提取的数值）。因此，现在的问题是如何使一个图像转换为一个数组，幸好现在有现成的软件可以实现这一功能，例如使用 bmp2h.exe 软件就可以实现。

13.4.1 如何将图片转换为 C 语言数组

因为 LCD 的分辨率是 320 像素×240 像素，因此，这就要求所要显示的图片具有如图 13-23 所示的属性。

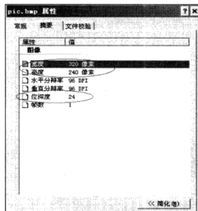


图 13-23 图片属性

下面的问题是：如何得到这么一幅图片呢？为了讲述方便，下面使用 Windows 自带的画图软件制作一个满足上述要求的图片。制作步骤如下。

（1）依次单击：开始\程序\附件\画图，打开画图软件后，选择图像\属性，将宽度和高度改为 320 和 240，如图 13-24 所示。

（2）在绘图区域，随便绘制几个字符或者形状，如图 13-25 所示。

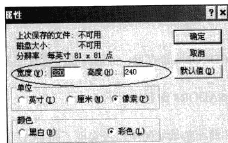


图 13-24 设置图片分辨率

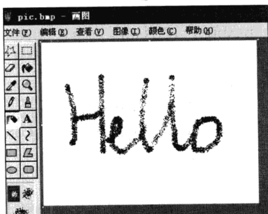


图 13-25 绘制图片

（3）选择文件\保存，在弹出的“保存为”对话框中输入图片名。注意，要用英文，如 pic，保存类型选择“24 位位图”，如图 13-26 所示。

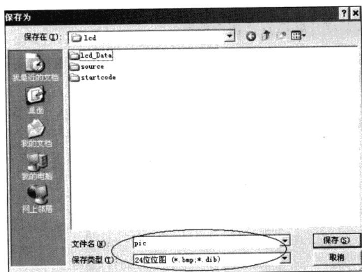


图 13-26 输入图片名

下面的工作是，使用 bmp2h.exe 软件，将刚才制作的图片转换为对应的 C 语言数组。

（1）打开 bmp2h.exe 软件，单击“添加”按钮，如图 13-27 所示。

（2）在弹出的对话框中，打开刚才建立的图片 pic.bmp，如图 13-28 所示。

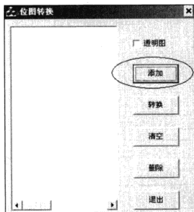


图 13-27　单击"添加"按钮

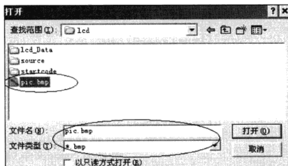


图 13-28　打开图片 pic.bmp

（3）单击"转换"按钮，即可在 pic.bmp 所在文件夹下生成对应的 C 语言数组文件 pic.h 和 pic.c 文件，如图 13-29 所示。

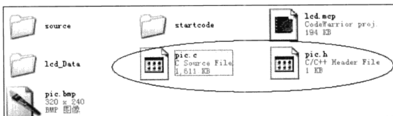


图 13-29　生成的 pic.h 和 pic.c 文件

（4）此时，需要将 pic.h 和 pic.c 文件添加到该工程中，如图 13-30 所示。

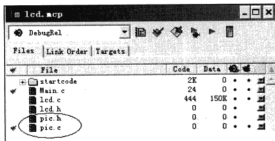


图 13-30　添加 pic.h 和 pic.c 文件

（5）打开 pic.c 文件，如图 13-31 所示，将矩形框中的内容删除，然后，将#include "base.h" 修改为#include "pic.h"，pic.c 文件中的其他内容不需要改动。

修改后，pic.c 文件内容如图 13-32 所示。

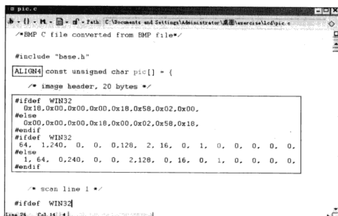


图 13-31 pic.c 文件（修改前）

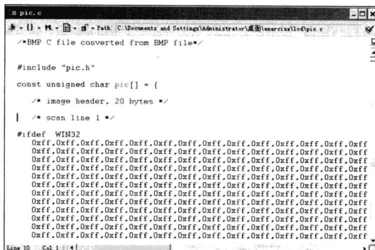


图 13-32 pic.c 文件（修改后）

13.4.2 程序代码分析

图片显示实验的文件布局如图 13-33 所示。

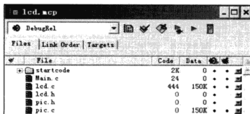


图 13-33 图片显示实验的文件布局

本实验主要是对上述实验的修改，在上述实验中讲解过的知识不再赘述。

LCD 模块包含两个文件：lcd.h 和 lcd.c 文件。lcd.h 文件内容没有变化。

lcd.c 文件内容如下：

```
#include "2440addr.h"
#include "lcd.h"

#define LOW21BITS(n)      ((n) & 0x1fffff)
#define Lcd_Enable()      rLCDCON1 |= 1
volatile unsigned short LCD_BUFFER[240][320];

static void Lcd_Config(void)
{
rGPCCON = 0xaaaa02a9;
rGPDCON = 0xaaaaaaaa;

rLCDCON1 = (CLKVAL_TFT << 8)|(3 <<5 )|(BPPMODE_TFT << 1) ;

rLCDCON2 = (VBPD << 24)|(LINEVAL_TFT << 14)|(VFPD << 6)|(VSPW);
rLCDCON3 = (HBPD << 19)|(HOZVAL_TFT << 8)|(HFPD);
rLCDCON4 |= (HSPW);
rLCDCON5 = (FRM565_TFT << 11) | (INVVCLK_TFT << 10) |
                    (INVVLINE_TFT << 9) | (INVVFRAME_TFT << 8) | (HWSWP);

 rLCDSADDR1 = (((unsigned int)LCD_BUFFER >> 22) << 21) |
LOW21BITS((unsigned int)LCD_BUFFER >> 1);
 rLCDSADDR2 = LOW21BITS( ((unsigned int)LCD_BUFFER +
(LCD_YSIZE_TFT * LCD_XSIZE_TFT *2)) ) >> 1 );
 rLCDSADDR3 = (0 << 11) | (LCD_XSIZE_TFT / 1);
}
static void Lcd_PowerEnable(int powerEnable)
{

  rGPGCON = rGPGCON & (~(3<<8)) |(3<<8);
  rGPGDAT = rGPGDAT | (1<<4) ;

  rLCDCON5 = rLCDCON5 & (~(1<<3)) | (powerEnable<<3);

}
```

这两个函数在前文中已经进行了详细讲解，在此不再赘述。

```
void Paint_Bmp(const unsigned char bmp[])
{
```

```
    int x,y;
    unsigned short c;
    int p = 0;

1    for( y = 0 ; y < 240 ; y++ )
     {
2        for( x = 0 ; x < 320 ; x++ )
         {
3                c = bmp[p+1]  |  (bmp[p]<<8) ;
4                if ( ( x < 320) && ( y < 240) )
5                    LCD_BUFFER[y][x] = c ;
6                p = p + 2 ;
         }
     }
}
```

这是显示一幅图片的函数。

第 1～2 行使用两层 for 循环，对 LCD_BUFFER[240][320]中的每个元素赋值。

第 3 行，因为图片生成的 C 语言数组是 unsigned char 型，而 LCD 的显示模式是 16 BPP，即 unsigned short 型，所以需要转换。

第 4 行，判断显示范围。

第 5 行，将该像素的数据写入上述数组的对应位置即可。

第 6 行，调整下标，继续读取下一个像素数据。

```
void Lcd_Init(void)
{
    Lcd_Config() ;
    Lcd_Enable() ;
    Lcd_PowerEnable(1);
}
```

Main.c 文件内容如下：

```
#include "lcd.h"
#include "pic.h"
int Main()
{
    Lcd_Init() ; ;
    while(1)
    {
        Paint_Bmp(pic) ;
    }
    return 0;
}
```

13.4.3 实例测试

将上述工程编译，生成.bin格式的二进制文件，将其下载到NAND FLASH中，启动开发板，可以看到LCD上已经显示了对应的图片，如图13-34所示。



图13-34 LCD显示效果

到此为止，实现了一幅图片在TFT LCD上的显示。读者可以自行使用其他的图片来显示。这也是制作电子相册的基础，电子相册制作将在第15章“综合实战”中进行讲解。

13.5 LCD高级实验之汉字显示

经过前面的讲解，基本的图片显示问题已经解决。现在讲解汉字显示的问题。在前文讲述过程中采用了从简单到复杂的讲述方法，从显示一个像素开始，逐步扩展，最后成功显示一幅图片。

类似的，下面从最简单的入手，首先学习汉字显示的基本原理，然后集中精力显示一个汉字，最后实现多行汉字的显示。

13.5.1 两种常见的汉字编码

与汉字有关的编码主要有：区位码和机内码。

1．区位码

1981年，中国国家标准总局发布了GB 2312—80《信息交换用汉字编码字符集・基本集》，也称为GB0。该文件将所有的国标汉字及符号分配在一个94行×94列的方阵中。

方阵的每一行称为一个“区”，编号为01区到94区。

方阵的每一列称为一个“位”，编号为01位到94位。

将上述方阵中的每个汉字或符号所在的区号和位号组合在一起，就是所谓的“区位码”。区位码的前两位是它的区号，后两位是它的位号。用区位码就可以唯一地确定一个汉字或符号，同理，任何一个汉字或符号也都对应着一个唯一的区位码。

例如，汉字“中”字的区位码是5448，表明它在方阵的54区48位。汉字“华”字的区位码是2710，表明它在方阵的27区10位。

一般，汉字点阵字库是根据区位码的顺序进行存储的，因此，可以根据区位来获取一个字库的点阵，计算公式如下：

点阵起始位置=((区码−1)×94+(位码−1))×汉字点阵字节数

获取点阵起始位置后，就可以从这个位置开始，读取出一个汉字的点阵，将其显示在LCD上。

2. 机内码

在计算机中，以ASCII码的形式存储字符数据，但是存储汉字时采用的是机内码。

机内码与区位码之间可以相互转化。区位码分为区码和位码，其取值均在1～94之间，如直接用区位码作为机内码，就会与基本ASCII码混淆。为了避免机内码与基本ASCII码的冲突，需要避开基本ASCII码。采用的方法是：先将区码和位码分别加上20 H，在此基础上再加80 H（此处“H”表示前两位数字为十六进制数）。

用机内码表示一个汉字需要使用两个字节，分别称为机内码高位字节和机内码低位字节，这两位字节的机内码的编码规则如下：

机内码高位字节 ＝ 区码 ＋20 H＋80 H＝ 区码 ＋A0 H

机内码低位字节 ＝ 位码 ＋20 H＋80 H＝ 位码 ＋A0 H

由于汉字的区码与位码的取值范围1～94，即0x01～0x5E，所以汉字区位码的高位字节与低位字节的取值范围均为0xA1～0x FE。

例如，汉字“啊”的区位码为1601，区码和位码分别用十六进制数表示即为1001 H，则它的机内码的高位字节为B0 H，低位字节为A1 H，机内码就是B0A1 H。

小技巧：如果读者对上面讲述的编码知识不熟悉，可以略过此部分，直接做后面的实验，通过实验，争取对汉字显示有个直观的体验，然后再学习这部分知识。

13.5.2 LCD汉字显示原理

汉字显示从本质上说就是根据汉字的区位码到字库中查找到该汉字的点阵数据，然后将其写入到帧内存即可。

所谓的汉字字库就是以汉字的区位码为索引的数组。下面讲解过程中实验的汉字点阵是16×16点阵，即每个汉字需要使用256个二进制位表示，即32个字节。因此，对于16×16点阵模式的字库，某个汉字点阵数据的起始地址计算公式为：

点阵起始位置=((区码−1)×94+(位码−1))×32

下面讲解汉字显示函数。

```
1    extern unsigned char __CHS[];
void Lcd_Print_ZW(unsigned int x,unsigned int y,unsigned short int QW,unsigned int c)
{
        unsigned short int i,j;
        unsigned char *pZK,mask,buf;
2       pZK = &__CHS[ (  ( (QW >> 8) - 1 )*94 + (QW & 0x00FF)- 1 )*32 ];
        for( i = 0 ; i < 16 ; i++ )
        {
```

```
3               mask = 0x80;
4               buf = pZK[i*2];
5               for( j = 0 ; j < 8 ; j++ )
                {
6                    if( buf & mask )
7                    {
8                        PutPixel(x+j,y+i,c);
                     }
9                    mask = mask >> 1;
                }

10              mask = 0x80;
11              buf = pZK[i*2 + 1];
12              for( j = 0 ; j < 8 ; j++ )
                {
13                   if( buf & mask )
                     {
14                       PutPixel(x+j + 8,y+i,c);
                     }
15                   mask = mask >> 1;
                }
        }
}
```

该函数的前两个参数是显示汉字时的坐标值，第 3 个参数是该汉字的区位码，第 4 个参数是所要显示的颜色。关键是第 3 个参数如何填写，下面举个具体的例子说明。

现在已知汉字“中”的区位码为 5448，则需要使用两个字节来表示区位码，高字节表示区位码的前两位，低字节表示区位码的后两位，即 54 << 8 + 48。

但是，这个地方错了！不知道读者能不能自己发现错误。这也是一些公司笔试题目中的一个常见的考点。因为左移运算符的优先级要低于加法运算符号，所以该参数正确的计算方法为：(54 << 8)+48。在后面实验时，读者不妨自己试一试，比较两种方式的区别。

第 1 行，声明了外部字库数组（读者可以从本书光盘中该实验文件夹下找到该字库文件 Font_libs.c）。

第 2 行，根据区位码查找所要显示汉字的点阵数据的首地址。

第 3～9 行，将偶数行显示，基本思路是根据点阵数据的值来判断是否显示该像素。如果该位为 1，则显示该像素；如果该位为 0，则不显示该像素。这样，汉字的点阵数据和 LCD 屏幕上的显示像素建立了对应关系。

第 10～15 行，将奇数行显示。

13.5.3 程序代码分析

汉字显示实验的文件布局如图 13-35 所示。

本实验主要是对上述实验的修改，在上述实验中讲解过的知识不再赘述。

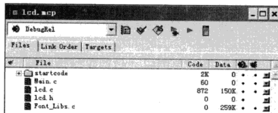


图 13-35　汉字显示实验的文件布局

LCD 模块包含两个文件：lcd.h 和 lcd.c 文件，lcd.h 文件内容没有变化。lcd.c 文件内容如下：

```
#include "2440addr.h"
#include "lcd.h"
#define LOW21BITS(n)((n) & 0x1fffff)        // To get lower 21bits
#define Lcd_Enable()  rLCDCON1 |= 1

volatile unsigned short LCD_BUFFER[240][320];
extern unsigned char __CHS[];

static void Lcd_Config(void)
{
        rGPCCON = 0xaaaa02a9;
        rGPDCON = 0xaaaaaaaa;

        rLCDCON1 = (CLKVAL_TFT << 8)|(3 <<5 )|(BPPMODE_TFT << 1)   ;
        rLCDCON2 = (VBPD << 24)|(LINEVAL_TFT << 14)|(VFPD << 6)|(VSPW);
        rLCDCON3 = (HBPD << 19)|(HOZVAL_TFT << 8)|(HFPD);
        rLCDCON4 = (HSPW);
        rLCDCON5 = (FRM565_TFT << 11) | (INVVCLK_TFT << 10) |
                   (INVVLINE_TFT << 9) | (INVVFRAME_TFT << 8) | (HWSWP) ;
        rLCDSADDR1 = (((unsigned int)LCD_BUFFER >> 22) << 21) |
                     LOW21BITS((unsigned int)LCD_BUFFER >> 1);
        rLCDSADDR2 = LOW21BITS( ((unsigned int)LCD_BUFFER +
                  (LCD_YSIZE_TFT * LCD_XSIZE_TFT *2) ) >> 1 );
}

static void Lcd_PowerEnable(int powerEnable)
{

    rGPGCON = rGPGCON & (~(3<<8)) |(3<<8);
    rGPGDAT = rGPGDAT | (1<<4) ;

    rLCDCON5 = rLCDCON5 & (~(1<<3)) | (powerEnable<<3);
```

```
}

static void PutPixel(unsigned int x,unsigned int y, unsigned short c )
{
    if ( (x < 320) && (y < 240) )
        LCD_BUFFER[(y)][(x)] = c;
}

void Lcd_ClearScr( unsigned int c)
{
    unsigned int x,y ;

    for( y = 0 ; y < 240 ; y++ )
    {
        for( x = 0 ; x < 320 ; x++ )
        {
            LCD_BUFFER[y][x] = c ;
        }
    }
}
```

上述 4 个函数在基础实验部分中已经进行了讲解，在此不再赘述。

```
void Lcd_Print_ZW(unsigned int x,unsigned int y,unsigned short int QW,unsigned int c)
{
        unsigned short int i,j;
        unsigned char *pZK,mask,buf;

        pZK = &__CHS[ (  ((QW >> 8) - 1 )*94 + (QW & 0x00FF)- 1 )*32 ];
        for( i = 0 ; i < 16 ; i++ )
        {
            mask = 0x80;
            buf = pZK[i*2];
            for( j = 0 ; j < 8 ; j++ )
            {
                if( buf & mask )
                {
                    PutPixel(x+j,y+i,c);
                }
                mask = mask >> 1;
            }

            mask = 0x80;
```

```
                buf = pZK[i*2 + 1];
                for( j = 0 ; j < 8 ; j++ )
                {
                        if( buf & mask )
                        {
                                PutPixel(x+j + 8,y+i,c);
                        }
                        mask = mask >> 1;
                }
        }
}
```

以上函数是在 LCD 上显示一个汉字的函数，在上文中已经进行了讲解。

```
void Lcd_Init(void)
{
    Lcd_Config() ;
    Lcd_Enable() ;
    Lcd_PowerEnable(1);
}
```

Main.c 文件内容如下：

```
#include "lcd.h"

int Main()
{
    Lcd_Init() ;
    Lcd_ClearScr(0xFFFF) ;
    while(1)
    {
        Lcd_Print_ZW(100,60,(54 << 8) + 48,0) ;
        Lcd_Print_ZW(100,80,(27 << 8) + 10,0) ;
    }
    return 0;
}
```

首先初始化 LCD，然后调用 Lcd_ClearSrc()函数，使整个屏幕显示白颜色。

在主循环中，调用汉字显示函数，显示“中”和“华”，尤其需要注意 Lcd_Print_ZW() 函数第 3 个参数是如何形成的。

13.5.4 实例测试

将上述工程编译，生成.bin 格式的二进制文件，将其下载到 NAND FLASH 中，启动开发板，可以看到 LCD 上已经显示了“中”、“华”两个汉字，如图 13-36 所示。

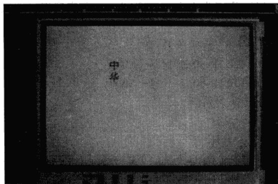


图 13-36　汉字显示实验测试

小技巧：前文讲到函数 Lcd_Print_ZW(unsigned int x,unsigned int y,unsigned short int QW,unsigned int c)第 3 个参数在赋值时需要特别注意，读者可以将 Main.c 文件中如下两句

```
Lcd_Print_ZW(100,60,(54 << 8) + 48,0) ;
Lcd_Print_ZW(100,80,(27 << 8) + 10,0) ;
```

修改为

```
Lcd_Print_ZW(100,60,54 << 8 + 48,0) ;
Lcd_Print_ZW(100,80,27 << 8 + 10,0) ;
```

观察修改后，还能不能正确地显示“中”、“华”两个汉字。这也提醒读者注意，移位运算符的优先升级低于加法运算符，这也是很多公司招聘笔试试题中的一个常见考点。

13.5.5　LCD 显示高级技巧——可变参函数 Lcd_Printf 的实现

在前面的实验中，成功地显示了汉字。但是，上述汉字显示函数 Lcd_Print_ZW(unsigned int x,unsigned int y,unsigned short int QW,unsigned int c)使用起来很不方便（需要知道所要显示的汉字的区位码），下面对其改进，使其能够较为灵活地使用。

例如，改进后可以使用如下方式：

```
Lcd_Printf(34,130,562, " 汉字显示示实验 ");
Lcd_Printf(34,130,562,"汉字实验 ");
```

该函数的原型为：

```
void Lcd_Printf(unsigned int x,unsigned int y,unsigned int c,char *fmt,...)
```

前两个参数是显示坐标，第 3 个参数是颜色值，第 4 个参数是可变参数（可变参数函数在第 10 章中曾经使用过该方法实现 UART 打印函数的编写）。

汉字字符串"汉字显示示实验 "中的每个汉字都是以机内码的形式存放（这里的道理就像字符串"hello "中的每个字母都是以 ASCII 码（American Standard Code for Information Interchange，美国信息互换标准代码）的形式存放一样），因此，显示的时候只需要将机内

码转换成区位码，然后以区位码为索引在汉字字库中查找到相应的汉字。现在的主要问题是如何将机内码转换为区位码。

前文讲到区位码和机内码的转换关系为：

机内码高位字节=区码+20 H+80 H=区码+A0 H

机内码低位字节=位码+20 H+80 H=位码+A0 H

所以，可以得到如下转换关系：

区码=机内码高位字节−A0 H

位码=机内码低位字节−A0 H

这样就完成了机内码到区位码的转换。下面讲解 Lcd_Printf(unsigned int x,unsigned int y,unsigned int c,char *fmt,...)函数的具体实现。

```
void Lcd_Printf(unsigned int x,unsigned int y,unsigned int c,char *fmt,...)
{
     unsigned char LCD_Printf_Buf[256];
1    va_list ap;
     unsigned char *pStr = LCD_Printf_Buf;
     unsigned int i = 0;

2    va_start(ap,fmt);
3    vsprintf(LCD_Printf_Buf,fmt,ap);
4    va_end(ap);
5    while(*pStr != 0 )
     {
6        switch(*pStr)
         {
             case '\n' :
                     break;
             default:
               {
7                  if( *pStr > 0xA0 & *(pStr+1) > 0xA0 )
                   {
8                      Lcd_Print_ZW( x , y , ((*pStr - 0xA0) << 8) + *(pStr+1) - 0xA0 , c );
                       pStr++;
                       x += 16;
                   }
                 break;
               }
         }
     }
}
```

可变参数函数的参数列表分为两部分：固定参数和个数可变的可变参数。函数中至少有一个固定参数。可变参数由于个数不确定，声明时用"..."表示。

- va_list ap：定义了一个指向可变参数列表指针。
- va_start(ap, argN)：使参数列表指针 ap 指向函数参数列表中的第一个可变参数，argN 是最后一个固定参数。例如，当函数的声明是 void va_test(char a, char b, …)时，它的固定参数依次是 a，b，最后一个固定参数 argN 为 c，因此就是 a_start(ap, c)。
- va_end(ap)：清空参数列表，并置参数指针 ap 无效，该宏的作用是结束可变参数的获取。
- vsprintf()函数原型为 int vsprintf(char *string, char *format, va_list param)，其作用是将 param 按格式 format 写入字符串 string 中。

因此上述函数的基本流程是：

（1）开辟一块区域存储可变参数。

（2）调用 vsprintf()函数将可变参数按照指定的格式复制到缓冲区 LCD_Printf_Buf[256]中。

（3）调用 Lcd_Print_ZW ()函数将该缓冲区中的汉字一个一个地显示出来。

第 1～5 行，实现的是将可变参数部分的数据复制到缓冲区 LCD_Printf_Buf[256]中。

因为 pStr 指针指向了该缓冲区，又因为字符串中的汉字是以机内码的形式存放的，所以，*pStr 指向的是该汉字机内码的高位，*（pStr+1）指向的是该汉字机内码的低位，即此时：

区码 = *pStr - 0xA0

位码 = *(pStr+1) - 0xA0

在显示一个汉字实验中，已知“中”的区位码是 5448，然后调用汉字显示函数 Lcd_Print_ZW(100,60,(54 << 8) + 48,0)，其中第三个参数的传递需要注意。可以使用如下方式调用：Lcd_Print_ZW(x , y , ((*pStr - 0xA0) << 8) + *(pStr+1) - 0xA0 , c)，如第 8 行。

将该函数添加到 lcd.c 文件中，然后就可以在 Main.c 文件中调用。

注意：本函数只是为了展示显示汉字的原理论，对于显示的数据不是汉字时怎么办等其他情况没有处理。请读者注意，只有先成功显示了，然后才考虑异常情况的处理。在很多情况下，初学者容易本末倒置，被一些莫名其妙的异常处理弄得一头雾水。任何一个初学者，在没有足够的开发经验时，刻意地强调函数的全面性是没有意义的，只有在开发过程中确实遇到了问题，才会试图去寻找增强函数健壮性的方法。

Main.c 文件内容如下：

```
#include "lcd.h"

int Main()
{
    Lcd_Init() ;
    Lcd_ClearScr(0xFFFF) ;
    while(1)
    {
        Lcd_Printf(34,130,0,"汉字显示示实验\n");
        Lcd_Printf(80,50,0,"汉字显示示实验\n");
    }
    return 0;
}
```

13.5.6 可变参函数 Lcd_Printf 测试

将上述工程编译，生成.bin 格式的二进制文件，将其下载到 NAND FLASH 中，启动开发板，可以看到 LCD 上已经显示了相应的汉字，如图 13-37 所示。

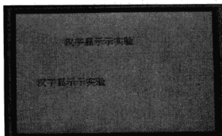


图 13-37 Lcd_Printf 测试结果

13.5.7 汉字区位码的思考

前文讲到，字母是以 ASCII 码（American Standard Code for Information Interchange，美国信息互换标准代码）的形式存储，汉字是以机内码的形式存储（由机内码可以计算得到区位码），但是在初学阶段，如果对这部分知识没有直观的概念，则很难理解汉字的显示。为了帮助初学者建立感性认识，下面设计了一个小实验。

本实验的功能：查看字符串“A 中华”是怎么存储的。

大写字母“A”的 ASCII 码为 0x41，“中华”两个汉字的区位码分别为 5448、2710，将区位码转换为十六进制数的形式，即

中：5448 ----> (54<<8)+48 -------> 0x3630

华：2710 ----> (27<<8)+10 -------> 0x1B0A

再结合前文讲到区位码和机内码的转换关系为：

机内码高位字节 = 区码 + 20 H + 80 H = 区码 + A0 H

机内码低位字节 = 位码 + 20 H + 80 H = 位码 + A0 H

因此可得到汉字对应的机内码的对应关系，如表 13-4 所示。

表 13-4 汉字对应的机内码的对应关系

字　符	存储方式		十六进制数
A	ASCII 码		0x41
中	机内码	高位	0xD6=0x36+0xA0
		低位	0xD0=0x30+0xA0
华	机内码	高位	0xBB=0x1B+0xA0
		低位	0xAA=0x0A+0xA0

请读者注意，上述过程是从理论上分析机内码的产生过程，但是，汉字存储时本身就是存储的机内码，所以字符串“A 中华”在内存中的存储方式如图 13-38 所示。

因此，本实验所采用的方法是从该字符串的首地址开始读取 5 个字节，然后以十六进制数的形式输出到串口。如果上面推导过程无误，读到的数据依次为：0x41、0xD6、0xD0、0xBB、0xAA。

本工程文件布局如图 13-39 所示。

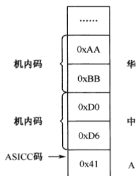


图 13-38 字符串“A 中华”在内存中的存储方式

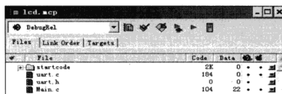


图 13-39 工程文件布局

UART 模块包含两个文件：uart.h 和 uart.c 文件。

uart.h 文件内容如下：

```
#ifndef __UART_H__
#define __UART_H__

extern void Uart0_Init(unsigned int baudrate) ;
extern void putc(unsigned char c) ;

#endif
```

uart.c 文件内容如下：

```
#include "2440addr.h"
#include "uart.h"

#define TXD0READY (1 << 2)
#define PCLK 50000000//时钟源设为 PCLK


void Uart0_Init(unsigned int baudrate)
{
      rGPHCON &= ~((3 << 4) | (3 << 6)) ;//GPH2-GPH3 是 RX/TX
      rGPHCON |= ((2 << 4) | (2 << 6)) ;//GPH2--TXD[0];GPH3--RXD[0]

      rGPHUP = 0x00 ;
```

```
        rULCON0 |= 0x03 ;                          //8 个数据位，1 个停止位
        rUCON0 = 0x05 ;
        rUBRDIV0    = PCLK / baudrate / 16 - 1 ;
        rURXH0 = 0;
}

void putc(unsigned char c)
{

        rUTXH0 = c ;
        while(!(rUTRSTAT0 & TXD0READY)) ;//等待上个字符发送完毕
}
```

主要是实现 UART 的初始化已经输出一个字符，上述函数在第 10 章中已做过讲解，在此不做赘述。

Main.c 文件内容如下：

```
#include "uart.h"
unsigned char table[]={0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,
                                    0x38,0x39,0x41,0x42,0x43,0x44,0x45,0x46} ;

int Main()
{
        unsigned char str[] = "A 中华" ;
        unsigned char *pStr ;
        unsigned int i;
        pStr = str ;
        Uart0_Init(115200) ;

                for(i = 0 ; i < 5 ; i++)
                {
                        putc(table[*(pStr+i)/16]) ;
                        putc(table[*(pStr+i)%16]) ;
                        putc(' ') ;
                }
                 return 0;
}
```

上述函数中使用 for 循环依次输出 pStr 指向的字符，而 pStr 指向了字符串"A 中华"的首地址，因此就可输出该字符串在内存中存储的数据。

13.5.8 实例测试

将上述工程编译，生成.bin 格式的二进制文件，将其下载到 NAND FLASH 中，打开超级终端，波特率设为 115 200，启动开发板，可以看到超级终端已经输出了串口发送的数据，

如图 13-40 所示。

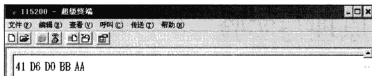


图 13-40　超级终端输出

可见，输出结果与上面的推导是一致的，这也在此验证了这一结论：字母以 ASCII 码的形式存储，汉字以机内码的形式存储！但是，汉字字库以区位码为索引，所以需要将汉字的机内码转换为区位码，然后以区位码为索引，在字库中查找到相应的汉字点阵数据。

笔者尽量使读者明白这样一种学习思路：从最简单的方法入手，尽量使问题简单化，先掌握总体的工作流程，对其他细节的东西，只在掌握了基本知识之后再考虑。

13.6　本章小结

本章主要讨论了 S3C2440 处理器 LCD 控制器的初始化以及使用 LCD 显示图片和汉字的原理，同时给出了简单的实验，帮助初学者将理论和实际结合起来。本章重在原理的讲述，力图帮助初学者尽快掌握 LCD 的使用技巧。

第 14 章

ADC 原理与实验

数据的采集、存储与显示是嵌入式系统的常见功能，在前面章节中已经对存储和显示进行了讲解。下面讲解数据的采集，一般对于模拟信号的采集是通过 ADC 来完成的。S3C2440 的 CMOS 模拟数字转换器 ADC 可以对 8 通道模拟输入信号进行循环检测。S3C2440 的 ADC 和触摸屏公用一个 ADC 转换器，所以学习 ADC 也是学习触摸屏的基础。

S3C2440 ADC 的主要特性如下。

- 分辨率：10 位。
- 最大转换速率：500 KSPS。
- 微分线性度误差：±1.0 LSB。
- 积分线性度误差：±2.0 LSB。
- 供电电压：3.3 V。
- 模拟输入电压范围：0～3.3 V。

14.1 ADC 原理

ADC 是一种将模拟信号转化为数字信号的方法，一般要经过采样、保持、量化、编码 4 个步骤。在实际电路中，有些过程是合并进行的，如采样和保持，量化和编码在转换过程中是同时实现的。由奈奎斯特采样定理可知，当采样频率大于模拟信号中最高频率的 2 倍时，采样值才能不失真地反映原来模拟信号。

主要技术指标如下。

- 分辨率

通常以输出二进制的位数表示分辨率的高低，一般位数越多，量化单位越小，对输入信号的分辨能力就越高。

例如，输入模拟电压的变化范围为 0～3.3 V、分辨率为 8 位时，可以分辨的最小模拟电压为 $3.3\ V/2^{12}=0.8\ mV$；而分辨率为 12 位时，可以分辨的最小模拟电压为 $5V/2^{12}\approx1.22\ mV$。

- 转换误差

它是指在零点和满度都校准以后，在整个转换范围内，分别测量各个数字量所对应的模拟输入电压实测范围与理论范围之间的偏差，取其中的最大偏差作为转换误差的指标。它通常以相对误差的形式出现，并以 LSB 为单位表示。

- 转换速度

完成一次模数转换所需要的时间称为转换时间。在大多数情况下，转换速度是转换时间的倒数。

ADC 的转换速度主要取决于转换电路的类型，并联比较型 ADC 的转换速度最高，逐次逼近型 ADC 次之，双积分型 ADC 转换速度最低。

S3C 2440 处理器 ADC 功能图如图 14-1 所示，其中，虚线框中是与触摸屏有关的功能模块，初学时可以不考虑，学完 ADC 基本实验后，再看触摸屏部分即可。

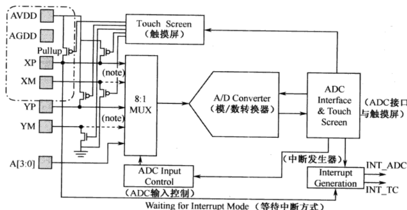


图 14-1　S3C 2440 处理器 ADC 功能图

从图 14-1 中可以看出，ADC 共有 8 路模拟输入，其中 XP、XM、YP 和 YM 是触摸屏使用的 4 路，剩下的 3 路模拟输入 A[0:3]可以用于一般的 ADC 输入通道。

此外，需要注意 ADC 的输入时钟是如何产生的。对于 S3C2440 处理器，ADC 输入时钟是由 PCLK 分频得到的，如图 14-2 所示。

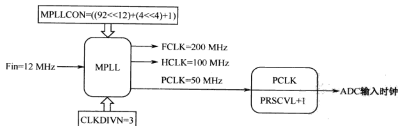


图 14-2　ADC 输入时钟的产生

注意：ADC 输入时钟不能超过 PCLK/5。

14.1.1　ADC 相关寄存器

使用 ADC 只需要对相应的寄存器进行配置，然后启动 ADC 即可，启动 ADC 有两种方法：

- 手动启动。
- 读取完上一次转换结果后自动启动下一次 ADC 转换。

得到 ADC 是否转换完成的信息有两种方法。

- 查询法：查询寄存器 ADCCON 的第 15 位（ADC 转换结束标志位）。
- 中断法：转换完成后，产生 ADC 中断信号，如图 14-1 中的 INT_ADC 信号。

当不使用触摸屏时，与 ADC 相关的寄存器主要有寄存器 ADCCON 和寄存器 ADCDAT0。寄存器 ADCCON 主要用于选择 ADC 的启动方式、设置 ADC 转换时钟以及 ADC 转换结束标志位等。寄存器 ADCDAT0 中存放了 ADC 转换所得到的数据，ADC 转换结束后，可以通过读该寄存器的值来得到转换结果。

寄存器 ADCCON 的各位含义如图 14-3 所示。

寄存器 ADCDAT0 的各位含义如图 14-4 所示。

ADCCON	位	功能描述
ECFLG	[15]	ADC 转换结束标志
		0：正在转换　　1：转换结束
PRSCEN	[14]	ADC 输入时钟是否分频
		0：不分频　　1：分频
PRSCVL	[13:6]	分频系数，取值范围为 0～255　ADC 时钟=PCLK/(PRSCVL+1)
		注意：ADC 时钟必须小于等于 PCLK/5
SEL_MUX	[5:3]	ADC 转换通道选择
		000：AIN0　001：AIN1　010：AIN2　011：AIN3
		100：YM　101：YP　110：XM　111：XP
STDBM	[2]	待机模式选择
		0：正常工作模式　1：待机模式
READ_START	[1]	读取上次转换结果后是否启动下次 AD 转换
		0：不启动　　1：启动
ENABLE_START	[0]	启动 ADC
		0：无效　　1：启动（启动后，会自动清零）

图 14-3　寄存器 ADCCON

ADCDAT0	位	功能描述
XPDATA	[0:9]	普通 ADC 转换数据值
		数据范围为：0～0x3FF

图 14-4　寄存器 ADCDAT0

14.1.2　ADC 初始化

对 ADC 初始化只需要做好以下两个方面的工作。

- 设置 ADC 输入时钟。
- 选择 ADC 输入通道。

可以使用如下代码初始化：

```
#define PRSCEN          1   //允许预分频
#define PRSCVL          49  //预分频值
#define STDBM           0   //正常工作模式
#define READ_START      0   //读数时不进行 A/D 转换

void ADC_Init(unsigned char channel)//unsigned char channel
{

    rADCCON   &= (~((1<<14)|(0xff<<6)|(0x7<<3)|(1<<2)|(1<<1)|(1<< 0)));
    rADCCON   |= (PRSCEN<<14)|( PRSCVL<<6)|(channel<<3)|(STDBM<<2)|
                 (READ_START<<1);
}
```

此时，对PCLK进行50分频，则可以计算出ADC输入时钟=PCLK/50=50 MHz/50=1 MHz。相信经过本书的学习，读者对上面初始化过程中使用“先与后或”的方式对寄存器进行初始化已经不陌生了。

14.2 ADC 实验

TQ2440 开发板上有一个可调电位器，电位器的中间抽头部分接在 ADC 输入通道 2 上，如图 14-5 所示，当电位器滑动头位于最下端时，AIN2 引脚电压为 0 V；当电位器滑动头位于最上端时，AIN2 引脚电压为 3.3 V；当电位器上、下滑动时，AIN2 引脚的电压值会在 0～3.3 V 之间变换。因此，本实验使用 ADC 输入通道 2 对 AIN2 引脚电压进行 A/D 转换，将取得的数字量输出到串口。

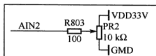


图 14-5 ADC 通道 2

14.2.1 ADC 实验代码详解

ADC 实验的文件布局如图 14-6 所示。

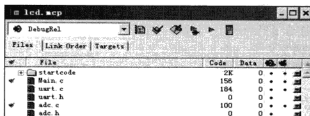


图 14-6 ADC 实验的文件布局

UART 模块包含两个文件：uart.h 和 uart.c 文件。

uart.h 文件内容如下：

```
#ifndef __UART_H__
```

```
#define __UART_H__

extern void Uart0_Init(unsigned int baudrate) ;
extern void putc(unsigned char c) ;

#endif
```

uart.c 文件内容如下：

```
#include "2440addr.h"
#include "uart.h"

#define TXD0READY (1 << 2)
#define RXD0READY (1 << 0)
#define PCLK 50000000
void Uart0_Init(unsigned int baudrate)
{
    rGPHCON &= ~((3 << 4) | (3 << 6)) ;
    rGPHCON |= ((2 << 4) | (2 << 6)) ;

    rGPHUP = 0x00 ;
    rULCON0 |= 0x03 ;
    rUCON0 = 0x05 ;
    rUBRDIV0    = PCLK / baudrate / 16 - 1 ;
    rURXH0 = 0;
}

void putc(unsigned char c)
{
    rUTXH0 = c ;
    while(!(rUTRSTAT0 & TXD0READY)) ;
}
```

上述函数在前面章节已经进行了讲解，在此不做赘述。

ADC 模块包含两个文件：adc.h 和 adc.c 文件。

adc.h 文件内容如下：

```
#ifndef __ADC_H__
#define __ADC_H__

extern int ADC_Read(void) ;
extern void ADC_Init(unsigned char channel) ;

#endif
```

adc.c 文件内容如下：

```
#include "2440addr.h"
#include "adc.h"

#define PRSCEN              1//允许预分频
#define PRSCVL              49//预分频值
#define STDBM               0//正常工作模式
#define READ_START          0//读数时不进行 A/D 转换
#define Adc_Start()    rADCCON |= 1
void ADC_Init(unsigned char channel)//unsigned char channel
{
    rADCCON   &=(~((1<<14)|(0xff<<6)|(0x7<<3)|(1<<2)|(1<<1)|(1<<0)));
    rADCCON    |= (PRSCEN<<14)|(PRSCVL<<6)|(channel<<3)|(STDBM<<2)|
                (READ_START<<1);
}
//ADC 初始化函数，手动启动 ADC，ADC 输入时钟为 1MHz。
int ADC_Read(void)
{

1    Adc_Start();
2    while(rADCCON  &  (1<<0)) ;//ADC 真正开始后，位[0]会自动清零
3    while(!(rADCCON  &  (1<<15)));

4    return ((int)(rADCDAT0  &  0x3ff));
}
```

第 1 行，启动 A/D 转换。

第 2 行，因为成功启动 A/D 转换后，该位会自动清零，因此在这里检查 ADC 是否真正启动。

第 3 行，使用查询方式等待 ADC 转换结束。

第 4 行，ADC 转换结束后，从寄存器 ADCDAT0 中读取 A/D 转换值。注意，低 10 位才是 A/D 转换值，所以需要将高位屏蔽掉。

Main.c 文件内容如下：

```
#include "adc.h"
#include "uart.h"

void IO_Init(void);
int Main(void)
{
    int value;
    IO_Init();
    while(1)
```

```
        {
1           value = ADC_Read();
2           putc(value/1000  +  '0');
3           putc(value%1000/100  +  '0');
4           putc(value%100/10  +  '0');
5           putc(value%10  +  '0');
        }
        return 0;
}
void IO_Init(void)
{
        Uart0_Init(115200);
        ADC_Init(2);
}
```

首先调用初始化函数将 UART 初始化，波特率为 115 200，然后初始化 ADC 输入通道 2。

第 1 行，启动 A/D 转换，并读取 A/D 转换值。

第 2～5 行，将 A/D 转换值发送到串口，后面加'0'是为了以十进制数的格式显示出来。

14.2.2 ADC 实验测试

编译、链接生成.bin 格式的二进制文件后下载到开发板，打开超级终端，波特率设为 115 200，这时会看到超级终端上显示出 A/D 转换的结果，如图 14-7 所示，调节开发板上的蓝色电位器，会发现该值在变化。

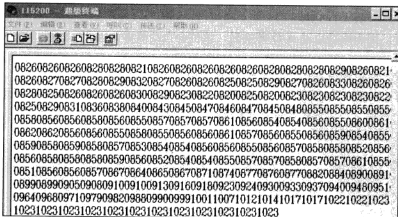


图 14-7　ADC 实验测试结果

在嵌入式开发过程中，简单的上位机软件设计也是需要了解的知识。下面使用电压表测试软件给读者展示上述数据采集结果，如图 14-8 所示。

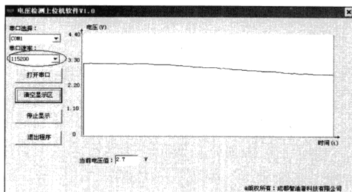


图 14-8　电压检测结果

14.3　本章小结

本章主要讨论了 S3C2440 处理器 ADC 的使用，ADC 在数据采集中使用比较普遍，同时触摸屏的各路信号采集也是以 ADC 为基础的，初学者可以自行学习触摸屏部分。

[illegible]

[illegible]

[illegible]

第 3 篇

典型项目分析

第15章 综合实战

15.1 实战1：数据采集系统实现

经过本书前面章节的学习，读者对各个功能模块已经有了整体的认识，下面结合具体的项目，向读者展示数据采集、存储与显示的整体过程。此外，特别练习结构体的使用。

15.1.1 功能描述

该实验功能是：使用ADC采集模拟电压数据，存储在NAND FLASH中，然后从NAND FLASH中读取所存储的电压值，显示在串口上。

15.1.2 模块划分

根据上述功能要求，可以从如下角度考虑系统模块的划分：

- 如何完成数据采集。
- 数据以何种形式存储，如何完成数据存储。
- 如何完成数据显示。

根据上述问题，结合模块化编程的方法，将具有独立功能的部分划分成一个个模块，每个模块包含两个文件：.h文件和.c文件。数据采集实验系统功能模块如图15-1所示。

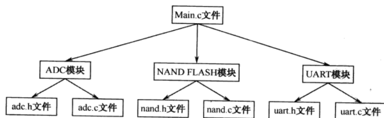


图15-1 数据采集实验系统功能模块

此外，还需要注意要采集的数据如何存储，因为这是一个简单的实验，因此规定数据存储格式如图15-2所示。

数 据 头	数 据	数 据 尾
14 字节	4 字节	14 字节

图 15-2 数据存储格式

注意：这里之所以要规定数据格式，主要是考虑到部分读者以后可能会学习网络编程、ZigBee无线传感器网络等方面的知识。这里的数据格式，也就是最简单的帧格式，在网络编程中，数据在各个层之间传递就是靠统一的数据结构来实现的，每层只负责处理解析本层所对应的部分。正因为有了统一的数据结构，才使得复杂的系统各个部分之间数据具有统一性，保证数据能顺利地传递。

此外，所谓的文件系统，也就是将数据的存储格式提前定义好，然后只提供接口参数，虽然存储介质不一样，但是接口都是统一的。

一般对于这种特定的数据存储方式，需要定义专门的结构体来访问，这也是本实验的基本目的之一。本实验中采用的数据结构是：

```
struct ADCVALUE
    {
        char head[14] ;
        unsigned char adcValue[4] ;
        char confirm[14] ;
    }
```

15.1.3 代码实现

数据采集实验的文件布局如图15-3所示。

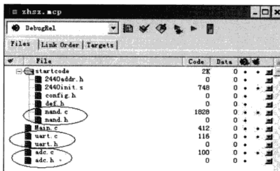


图 15-3 数据采集实验的文件布局

UART模块包含两个文件：uart.h和uart.c文件。

uart.h文件中声明了UART初始化函数Uart0_Init()。

```
#ifndef __UART_H__
#define __UART_H__
```

```
extern void Uart0_Init(unsigned int baudrate) ;
extern void putc(unsigned char c) ;
#endif
```

uart.c文件对上述三个函数进行了具体的实现。

```
#define PCLK 50000000              //时钟源设为 PCLK
void Uart0_Init(unsigned int baudrate)
{
1     rGPHCON &= ~((3 << 4) | (3 << 6)) ;
2     rGPHCON |= ((2 << 4) | (2 << 6)) ;
3     rGPHUP   = 0x00 ;
4     rULCON0 |= 0x03 ;                //8 个数据位，1 个停止位
5     rUCON0   = 0x05 ;
6     rUBRDIV0 = (int) (PCLK / baudrate / 16)- 1;
7     rURXH0    = 0;
}
```

第1～3行，将GPH2、GPH3配置为TXD、RXD模式。

第4行，设置寄存器ULCON0，设置数据发送格式为：8个数据位，1个停止位，无校验位。

第5行，发送模式和接收模式都使用查询方式。

第6行，设置波特率，其中波特率作为一个参数传递到该初始化函数。

第7行，将URXH0清零。

```
void putc(unsigned char c)
{

    rUTXH0 = c ;
    while(!(rUTRSTAT0 & (1 << 2))) ;//等待上一个字符发送完毕
}
```

该函数可以发送一个字符。

首先将要发送的字符存入寄存器UTXH0中，然后等待发送完毕，发送完毕后，寄存器UTRSTAT0的第2位会置1，然后跳出while循环。

NAND FLASH模块包含两个文件：nand.h和nand.c文件。

nand.h文件内容如下：

```
#ifndef __NAND_H
#define __NAND_H

1   #define CMD_READ1          0x00    //  Read1
2   #define CMD_READ2          0x30    //  Read2
3   #define CMD_READID         0x90    //  ReadID
4   #define CMD_WRITE1         0x80    //  Write phase 1
5   #define CMD_WRITE2         0x10    //  Write phase 2
```

```
6    #define CMD_ERASE1              0x60       //  Erase phase 1
7    #define CMD_ERASE2              0xd0       //  Erase phase 2
8    #define CMD_STATUS              0x70       //  Status read
9    #define CMD_RESET               0xff //     Reset

10   #define NF_Send_Cmd(cmd)        {rNFCMD   = (cmd); }
11   #define NF_Send_Addr(addr)      {rNFADDR = (addr); }
12   #define NF_Send_Data(data)      {rNFDATA8 = (data); }
13   #define NF_Enable()             {rNFCONT &= ~(1<<1); }
14   #define NF_Disable()            {rNFCONT |= (1<<1); }
15   #define NF_Enable_RB()          {rNFSTAT |= (1<<2); }        //开启 RnB 监视模式
16   #define NF_Check_Busy()         {while(!(rNFSTAT&(1<<2)));}
17   #define NF_Read_Byte()          (rNFDATA8)

18   #define TACLS                   1
19   #define TWRPH0                  4
20   #define TWRPH1                  0

21   extern void NF_Init(void) ;
22   extern void NF_ReadPage(unsigned int block,unsigned int page, unsigned char * dstaddr) ;
23   extern void NF_WritePage(unsigned int block,unsigned int page, unsigned char *buffer) ;
24   extern int   NF_EraseBlock(unsigned int block) ;//

#endif
```

第1～9行，定义了NAND FLASH的基本命令。

第10～17行，定义了NAND FLASH控制器的基本操作，包括NAND FLASH的开启关闭、检测忙信号，发送命令、地址、数据等操作。

注意：rNFDATA8是在2440addr.h文件中定义的，定义如下。

```
#define rNFDATA8 (*(volatile unsigned char *)0x4E000010)
```

第18～20行，定义了NAND FLASH的三个时序参数，具体分析见12.1.4节。

第21～24行，使用extern关键字声明了4个外部函数，这样就可以在其他文件中使用这几个函数。

nand.c文件内容如下：

```
#include "2440addr.h"
#include "Nand.h"

static void NF_Reset()
{
    NF_Enable();
    NF_Enable_RB();
    NF_Send_Cmd(CMD_RESET);
```

```
        NF_Check_Busy();
        NF_Disable();
}
void NF_Init(void)
{
        rGPACON &= ~(0X3F << 17);
        rGPACON |= (0X3F << 17);
        rNFCONF = (TACLS<<12)|(TWRPH0<<8)|(TWRPH1<<4);
        rNFCONT = (0<<12)|(1<<0);
        rNFSTAT = 0;
        NF_Reset();
}

void NF_ReadPage(unsigned int block,unsigned int page, unsigned char * dstaddr)
 {
        unsigned int i;
        unsigned int blockPage = (block<<6)+page;

        NF_Reset();
        NF_Enable();
        NF_Enable_RB();

        NF_Send_Cmd(CMD_READ1);          //CMD_READ1= 0x00

        NF_Send_Addr(0x00);
        NF_Send_Addr(0x00);
        NF_Send_Addr((blockPage) & 0xff);
        NF_Send_Addr((blockPage >> 8) & 0xff);
        NF_Send_Addr((blockPage >> 16) & 0x1);

        NF_Send_Cmd(CMD_READ2);       //CMD_READ12= 0x30

        NF_Check_Busy();

        for (i = 0; i < 2048; i++)
        {
             dstaddr[i] =   NF_Read_Byte();
        }

        NF_Disable();
}

void NF_WritePage(unsigned int block,unsigned int page, unsigned char *buffer)
```

```
{
    unsigned int i;
    unsigned int blockPage = (block<<6)+page;
    unsigned char *bufPt = buffer;

    NF_Reset();
    NF_Enable();    //控制器使能
    NF_Enable_RB(); //开启 RnB 监视模式

    NF_Send_Cmd(CMD_WRITE1); /* 写第一条命令 */

    NF_Send_Addr(0x00);
    NF_Send_Addr(0x00);
    NF_Send_Addr((blockPage) & 0xff);
    NF_Send_Addr((blockPage >> 8) & 0xff);
    NF_Send_Addr((blockPage >> 16) & 0x1);
    for(i=0;i<2048;i++)
    {
        NF_Send_Data(*bufPt++);  }

    NF_Send_Cmd(CMD_WRITE2);
    NF_Check_Busy();
    NF_Disable();
}
int NF_EraseBlock(unsigned int block)
{
    unsigned int blocknum=(block<<6);
    NF_Reset();
    NF_Enable();
    NF_Enable_RB();

    NF_Send_Cmd(CMD_ERASE1);
    NF_Send_Addr( blocknum & 0xff);
    NF_Send_Addr((blocknum>>8) & 0xff);
    NF_Send_Addr((blocknum>>16) & 0xff);

    NF_Send_Cmd(CMD_ERASE2) ;

    NF_Check_Busy() ;

    NF_Disable() ;
    return 1 ;
}
```

ADC模块包含两个文件：adc.h和adc.c文件。

adc.h文件内容如下：

```
#ifndef __ADC_H__
#define __ADC_H__

extern int ADC_Read(void) ;
extern void ADC_Init(unsigned char channel) ;

#endif
```

adc.c文件内容如下：

```
#include "2440addr.h"
#include "adc.h"

#define PRSCEN          1 //允许预分频
#define PRSCVL          49 //预分频值
#define STDBM           0 //正常工作模式
#define READ_START      0 //读数时不进行 A/D 转换
#define Adc_Start()     rADCCON |= 1


void ADC_Init(unsigned char channel)//unsigned char channel
{

    rADCCON &= (~((1 << 14) | (0xff << 6) | (0x7 << 3) |(1 << 2) | (1 << 1) | (1 << 0))) ;
    rADCCON |= (PRSCEN << 14) | (PRSCVL << 6) |(channel << 3)   |
                                    (STDBM << 2)|(READ_START << 1) ;

}
int ADC_Read(void)
{

    Adc_Start() ;

    while(rADCCON & (1 << 0)) ;//ADC 真正开始后，位[0]会自动清零
    while(!(rADCCON & (1 << 15))) ;
    return ((int)(rADCDAT0 & 0x3ff)) ;
}
```

上述函数在前面章节中已经进行了讲解，在此不再赘述。

Main.c文件内容如下：

```
#include "adc.h"
#include "uart.h"
```

```
#include "nand.h"
#include "string.h"
void IO_Init(void) ;

int Main(void)
{
1       struct ADCVALUE
        {
                char head[14] ;
                unsigned char adcValue[4] ;
                char confirm[14] ;
2       } valFormt[64] ;

        int i,j,value ;
3       struct ADCVALUE recBuf[64] ;
        IO_Init() ;
        while(1)
        {
4               for(i = 0 ; i < 64 ; i++ )
                {
5                       strcpy(valFormt[i].head,"The Value :      ") ;
6                       value = ADC_Read();
7                       value = 10 * value *3.3 / 1023 ;
8                       valFormt[i].adcValue[0] = (value/10 + '0') ;
9                       valFormt[i].adcValue[1] =    '.' ;
10                      valFormt[i].adcValue[2] = value%10 + '0' ;
11                      valFormt[i].adcValue[3] = 'V' ;
12                      strcpy(valFormt[i].confirm,"        test over! ") ;

                }
13              NF_EraseBlock(18) ;
14              NF_WritePage(18,5, (unsigned char *)valFormt) ;
15              NF_ReadPage(18,5, (unsigned char *)recBuf) ;
16              for(i = 0 ; i < 64 ; i++)
                {
17                      for(j = 0 ; j < 14 ; j++ )
                        {
18                              putc(recBuf[i].head[j]) ;
                        }
19                      for(j = 0 ; j < 4 ; j++ )
                        {
20                              putc(recBuf[i].adcValue[j]) ;
                        }
```

```
21              for(j = 0 ; j < 14 ; j++ )
                {
22                      putc(recBuf[i].confirm[j]) ;
                }
23              putc('\r') ;

        }
    }

    return 0;
}
void IO_Init(void)
{
    Uart0_Init(115200) ;
    ADC_Init(2) ;
    NF_Init() ;
}
```

第1～2行，声明了结构体ADCVALUE。注意，这个结构体是与数据的存储格式对应的，同时定义了一个结构体数组valFormt[64]，因为每帧数据占32个字节（一帧数据即满足图15-2规定的数据格式的数据集合），NAND FLASH中每页是2 KB，所以，每页最多可以存储64帧数据。

第3行，定义了一个结构体数组，从NAND FLASH读取数据时，存储在该结构体中。

第4行，循环次数是64次，每循环一次，进行一次A/D转换，同时将读取的数据按照规定的格式组成一帧数据。

第5行，给数据头赋值（这里的数据头是随便定义的，读者可以定义自己的数据头，但是在网络编程中，数据头是有特定格式的）。注意，字符串赋值需要借助于字符串复制函数，所以头文件中包含了#include "string.h"。

第7～10行，对转换后的数据进行处理，这里假定数据的显示格式类似于3.3 V、2.5 V等，所以需要将转换后的数据处理。下面通过一个例子来讲述数据处理的过程。

例：因为ADC转换精度为10位，所以，A/D转换最大值为1 023，即1 023表示输入电压为3.3 V，则读取到的A/D转换值为956时，对应的输入电压可以通过如下公式计算：956×3.3/1 023=3.08，但是一般处理小数不好处理，所以，将其放大10倍数，即956×3.3×10/1 023=30.8，然后对其处理即可，处理完后，数据在adcValue[4]中的存放形式（实际上是存储字符数据对应的ASCII码），如图15-4所示。

adcValue[0]	adcValue[1]	adcValue[2]	adcValue[3]
'3'	'.'	'0'	'V'

图 15-4　电压值存储形式

第12行，给数据尾赋值。注意，字符串赋值需要借助于字符串复制函数。

第13行，对NAND FLASH进行写操作前，需要进行擦除操作。

第14行，将采集的电压数据写入NAND FLASH的第18块中的第5页（读者可以随便设定一页）。注意，第3个参数的传递形式，需要强制类型转换。

第15行，从NAND FLASH的第18块中的第5页中读取电压数据，读取后的电压值存储在recBuf[64]中。

第16～22行，调用UART模块的字符输出函数putc()显示数据头、电压数据和数据尾。

15.1.4 实例测试

编译、链接生成.bin格式的二进制文件后下载到开发板，打开超级终端，波特率设为115200，调节开发板上蓝色可调电位器上方的旋钮，会在超级终端上显示出对应的电压值，如图15-5、图15-6、图15-7所示。

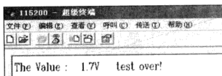


图 15-5 数据采集实验测试结果 1

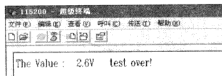


图 15-6 数据采集实验测试结果 2

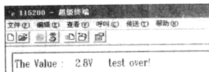


图 15-7 数据采集实验测试结果 3

15.1.5 实验总结

本实验主要向读者展示了数据采集、存储与显示系统的基本流程。在实际应用过程中，可能还会设计输入信号的放大、整形与滤波等知识，在此并没有进行讲解，读者需要根据所采集信号的特点进行分析。

15.2 实战 2：串口控制实验

在自动控制系统中，经常会遇到以下情况：上位机发送命令，下位机接收到命令后，根据不同的命令执行不同的动作。本节实验对上述控制系统进行模拟，通过超级终端发送命令，开发板收到命令后执行相应的操作。

15.2.1 功能描述

本实验主要是用于模拟自动控制系统中的上位机对下位机控制的情形。本实验通过串口发送命令，发送命令的格式以及下位机收到命令后执行的相关动作如图15-8所示。

串口发送的命令	开发板执行的操作
0	点亮 LED1
1	熄灭 LED1
2	在 LCD 上显示“控制命令接收成功！”
3	关闭 LCD

图 15-8　串口发送的命令和开发板执行的操作

15.2.2　模块划分

根据上述问题，结合模块化编程的方法，将具有独立功能的部分划分成一个个模块，每个模块包含两个文件（.h文件和.c文件），如图15-9所示。

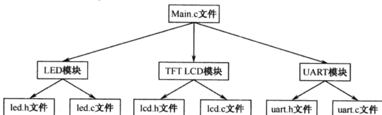


图 15-9　串口控制实验功能模块图

15.2.3　代码实现

串口控制实验的文件布局如图15-10所示，共分为三个模块（LED模块、LCD模块和UART模块），其中Font_Libs.c是字库文件，在第13章进行过讲解。

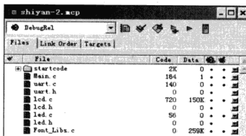


图 15-10　串口控制实验的文件布局

LED模块包含两个文件：led.h和led.c文件。

led.h文件内容如下：

```
#ifndef __LED_H__
#define __LED_H__

#include "2440addr.h"
```

```
#define Led1_On()        {rGPBDAT &= (~(1 << 5));}
#define Led1_Off()       {rGPBDAT |= (1 << 5);}

extern void Led_Init(void);

#endif
```

led.c文件内容如下：

```
#include "led.h"
#include "2440addr.h"


void Led_Init(void)
{
      rGPBCON &= ~(3 << 10);
      rGPBCON |= (1<<10) ;
      rGPBUP &= ~(1 << 5) ;
      rGPBDAT |= (1 <<5);
}
```

LCD模块包含两个文件：lcd.h和lcd.c文件。

lcd.h文件内容如下：

```
#ifndef __LCD_H__
#define __LCD_H__


#define MVAL              (0)
#define INVVDEN           (1)           //0=normal      1=inverted
#define HWSWP             (1)           //Half word swap control
#define PNRMODE           (3)           //设置为 TFT 屏
#define BPPMODE           (12)          //设置为 16BPP 模式

#define LCD_XSIZE_TFT     (320)
#define LCD_YSIZE_TFT     (240)
#define CLKVAL_TFT        (7)

#define VBPD          (14)
#define VFPD          (11)
#define VSPW          (2)

#define HBPD          (37)
#define HFPD          (19)
#define HSPW          (29)
```

```
#define HOZVAL_TFT       (320-1)
#define LINEVAL_TFT      (240-1)


#define BPPMODE_TFT                          (12)

#define FRM565_TFT                           (1)
#define INVVCLK_TFT                          (1)
#define INVVLINE_TFT                         (1)
#define INVVFRAME_TFT                        (1)

#define INVVD_TFT                            (0)
#define INVVDEN_TFT                          (0)
#define PWREN_TFT                            (0)

extern void Lcd_Init(void) ;
extern void Lcd_PowerEnable(int powerEnable) ;
extern void Lcd_Print_ZW(unsigned int x,unsigned int y,unsigned short int QW,unsigned int c) ;

#endif
```

lcd.c文件内容如下：

```
#include "2440addr.h"
#include "lcd.h"

#include <stdarg.h>

#define LOW21BITS(n)     ((n) & 0x1fffff)   // To get lower 21bits

#define Lcd_Enable()   rLCDCON1 |= 1

volatile unsigned short LCD_BUFFER[240][320];
extern unsigned char __CHS[];
static void Lcd_Config(void)
{
    rGPCCON = 0xaaaa02a9;
    rGPDCON = 0xaaaaaaaa;

    rLCDCON1 = (CLKVAL_TFT << 8)|(3 <<5 )|(BPPMODE_TFT << 1)   ;

    rLCDCON2 = (VBPD << 24)|(LINEVAL_TFT << 14)|(VFPD << 6)|(VSPW);
    rLCDCON3 = (HBPD << 19)|(HOZVAL_TFT << 8)|(HFPD);
```

```
    rLCDCON4 = (HSPW);
    rLCDCON5 = (FRM565_TFT << 11) | (INVVCLK_TFT << 10) |
                (INVVLINE_TFT << 9) | (INVVFRAME_TFT << 8) | (HWSWP) ;


    rLCDSADDR1 = (((unsigned int)LCD_BUFFER >> 22) << 21) |
                LOW21BITS((unsigned int)LCD_BUFFER >> 1);
    rLCDSADDR2 = LOW21BITS( ((unsigned int)LCD_BUFFER +
                (LCD_YSIZE_TFT * LCD_XSIZE_TFT *2) ) >> 1 );
    rLCDSADDR3 = (0 << 11) | (LCD_XSIZE_TFT / 1);

}

void Lcd_PowerEnable(int powerEnable)
{

    rGPGCON = rGPGCON & (~(3<<8)) |(3<<8);
    rGPGDAT = rGPGDAT | (1<<4) ;
    rLCDCON5 = rLCDCON5 & (~(1<<3)) | (powerEnable<<3);

}

void PutPixel(unsigned int x,unsigned int y, unsigned short c )
{
    if ( (x < 320) && (y < 240) )
        LCD_BUFFER[(y)][(x)] = c;
}

void Lcd_Print_ZW(unsigned int x,unsigned int y,unsigned short int QW,unsigned int c)
{
    unsigned short int i,j;
    unsigned char *pZK,mask,buf;

    pZK = &__CHS[ ( ( (QW >> 8) - 1 )*94 + (QW & 0x00FF)- 1 )*32 ];
    for( i = 0 ; i < 16 ; i++ )
    {

        mask = 0x80;
        buf = pZK[i*2];
        for( j = 0 ; j < 8 ; j++ )
        {
            if( buf & mask )
            {
```

```
                    PutPixel(x+j,y+i,c);
                }
                mask = mask >> 1;
            }
            mask = 0x80;
            buf = pZK[i*2 + 1];
            for( j = 0 ; j < 8 ; j++ )
            {
                if( buf & mask )
                {
                    PutPixel(x+j + 8,y+i,c);
                }
                mask = mask >> 1;
            }
        }
}
void Lcd_Printf(unsigned int x,unsigned int y,unsigned int c,char *fmt,...)
{
        unsigned char LCD_Printf_Buf[256];
        va_list ap;
        unsigned char *pStr = LCD_Printf_Buf;
        unsigned int i = 0;
        va_start(ap,fmt);
        vsprintf(LCD_Printf_Buf,fmt,ap);
        va_end(ap);
        while(*pStr != 0 )
        {
        switch(*pStr)
            {
                case '\n' :
                        break;
                default:
                  {
                        if( *pStr > 0xA0 & *(pStr+1) > 0xA0 )
                        {
                            Lcd_Print_ZW( x , y , ((*pStr - 0xA0) << 8) + *(pStr+1)-0xA0 , c );
                            pStr++;
                            x += 16;
                        }
                        break;
                  }
            }
        }
```

```
}

void Lcd_Init(void)
{
    Lcd_Config() ;
    Lcd_Enable() ;
    Lcd_PowerEnable(1);
}
```

上述函数在前面相关章节中已经进行了讲解，在此不做赘述。

UART模块包含两个文件：uart.h和uart.c文件。

uart.h文件内容如下：

```
#ifndef __UART_H__
#define __UART_H__

extern void Uart0_Init(unsigned int baudrate) ;
extern unsigned char getc(void) ;

#endif
```

uart.c文件内容如下：

```
#include "2440addr.h"
#include "uart.h"

#define TXD0READY (1 << 2)
#define RXD0READY (1 << 0)
#define PCLK 50000000
void Uart0_Init(unsigned int baudrate)
{
    rGPHCON &= ~((3 << 4) | (3 << 6)) ;
    rGPHCON |= ((2 << 4) | (2 << 6)) ;

    rGPHUP = 0x00 ;
    rULCON0 |= 0x03 ;                    //8 个数据位，1 个停止位
    rUCON0 = 0x05 ;
    rUBRDIV0 = PCLK / baudrate / 16 - 1 ;
    rURXH0 = 0;
}
unsigned char getc(void)
{
    unsigned char c ;
    while(!(rUTRSTAT0 & RXD0READY)) ;
    c = rURXH0 ;
```

```
        return c ;
}
```

Main.c文件内容如下：

```
#include "led.h"
#include "uart.h"
#include "lcd.h"

void IO_Init(void) ;
extern const unsigned char __CHS[] ;
int Main(void)
{
        IO_Init() ;
        while(1)
        {
                switch(getc())
                {
                        case '0' :
                                Led1_On() ;
                                break ;
                        case '1' :
                                Led1_Off() ;
                                break ;
                        case '2' :
                                Lcd_Printf(10,40,254,"控制命令接收成功！") ;
                                break ;
                        case '3' :
                                Lcd_PowerEnable(0) ;
                                break ;
                        default :
                               break ;
                }
        }
        return 0;
}
void IO_Init(void)
{
        Uart0_Init(115200) ;
        Led_Init() ;
        Lcd_Init() ;
}
```

主循环中使用switch/case语句判断接收到的字符，然后调用相应的处理函数即可。

15.2.4 实例测试

编译、链接生成.bin格式的二进制文件后下载到开发板，打开超级终端，波特率设为115200，在超级终端中发送0，开发板上的LED1点亮，发送1，LED1熄灭；发送2，此时LCD上显示了“控制命令接收成功！”；发送3，LCD关闭。

15.2.5 实验总结

本实验主要用于模拟自动控制系统中的上位机对下位机控制的情形，虽然控制命令较为简单，但是本实验主要是向读者展示这一控制流程。在实际应用过程中，读者应根据具体项目要求设计相应的控制程序。

15.3 实战 3：制作电子相册

本节实验，主要是练习定时器中断的使用，结合LCD显示图片功能，将二者结合起来制作一个电子相册。

15.3.1 功能描述

使用定时器中断功能，定时器每秒产生一次中断，中断处理程序调用显示图片的函数，更新显示的图片，即电子相册每秒钟显示一幅图片。

注意：显示的图片分辨率是320像素×240像素，使用bmp2h.exe软件将图片转换为对应的C语言数组，具体转换方法，请读者参考本书13.4.1节“如何将图片转换为C语言数组”。

15.3.2 模块划分

电子相册实验系统模块如图15-11所示。

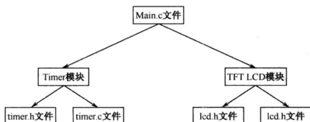


图 15-11 电子相册实验系统模块

15.3.3 代码实现

电子相册实验的文件布局如图15-12所示。

本实验有两个模块，LCD模块包含两个文件（lcd.h和lcd.c文件）；Timer模块包含6个文

件，其中timer.h和timer.c文件完成定时器的初始化，interrupt.h和interrupt.c文件完成定时器中断函数的初始化，isrservice.h和isrsercice.c文件完成定时器中断处理，pic.c和pic.h等文件是由图片生成的C语言数组文件。

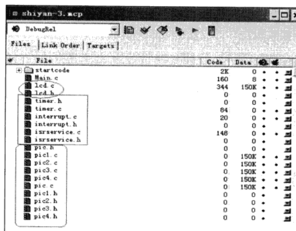


图 15-12　电子相册实验的文件布局

LCD模块包含两个文件：lcd.h和lcd.c文件。

lcd.h文件内容如下：

```
#ifndef __LCD_H__
#define __LCD_H__


#define MVAL            (0)
#define INVVDEN         (1)        //0=normal      1=inverted
#define HWSWP           (1)        //Half word swap control
#define PNRMODE         (3)        //设置为 TFT 屏
#define BPPMODE         (12)       //设置为 16BPP 模式

#define LCD_XSIZE_TFT   (320)
#define LCD_YSIZE_TFT   (240)

#define CLKVAL_TFT      (7)

#define VBPD            (14)
#define VFPD            (11)
#define VSPW            (2)

#define HBPD            (37)
#define HFPD            (19)
```

```
#define HSPW                    (29)

#define HOZVAL_TFT      (320-1)
#define LINEVAL_TFT     (240-1)

#define BPPMODE_TFT                     (12)
#define FRM565_TFT                      (1)
#define INVVCLK_TFT                     (1)
#define INVVLINE_TFT                    (1)
#define INVVFRAME_TF                    (1)

#define INVVD_TFT                       (0)
#define INVVDEN_TFT                     (0)
#define PWREN_TFT                       (0)

extern void Lcd_Init(void) ;
extern void Lcd_PowerEnable(int powerEnable) ;
extern void Paint_Bmp(const unsigned char bmp[]) ;

#endif
```

lcd.c文件内容如下：

```
#include "2440addr.h"
#include "lcd.h"

#define LOW21BITS(n)((n) & 0x1fffff)        // To get lower 21bits

#define Lcd_Enable()   rLCDCON1 |= 1

volatile unsigned short LCD_BUFFER[240][320];
extern unsigned char __CHS[];

static void Lcd_Config(void)
{
        rGPCCON=0xaaaa02a9;
        rGPDCON=0xaaaaaaaa;

        rLCDCON1=(CLKVAL_TFT << 8)|(3 <<5 )|(BPPMODE_TFT << 1)   ;
        rLCDCON2=(VBPD << 24)|(LINEVAL_TFT << 14)|(VFPD << 6)|(VSPW);
        rLCDCON3=(HBPD << 19)|(HOZVAL_TFT << 8)|(HFPD);
        rLCDCON4=(HSPW);
        rLCDCON5=(FRM565_TFT << 11) | (INVVCLK_TFT << 10) |
                (INVVLINE_TFT << 9) | (INVVFRAME_TFT << 8) | (HWSWP) ;
```

```
        rLCDSADDR1 = (((unsigned int)LCD_BUFFER >> 22) << 21) |
                    LOW21BITS((unsigned int)LCD_BUFFER >> 1);
        rLCDSADDR2 = LOW21BITS( ((unsigned int)LCD_BUFFER +
                    (LCD_YSIZE_TFT * LCD_XSIZE_TFT *2) ) >> 1 );
        rLCDSADDR3 = (0 << 11) | (LCD_XSIZE_TFT / 1);

}

void Lcd_PowerEnable(int powerEnable)
{

        rGPGCON = rGPGCON & (~(3<<8)) |(3<<8);
        rGPGDAT = rGPGDAT | (1<<4) ;

        rLCDCON5 = rLCDCON5 & (~(1<<3)) | (powerEnable<<3);

}

void PutPixel(unsigned int x,unsigned int y, unsigned short c )
{
        if ( (x < 320) && (y < 240) )
            LCD_BUFFER[(y)][(x)] = c;
}

void Paint_Bmp(const unsigned char bmp[])
{
        int x,y;
        unsigned short c;
        int p = 0;

        for( y = 0 ; y < 240 ; y++ )
        {
            for( x = 0 ; x < 320 ; x++ )
            {
                c = bmp[p+1] | (bmp[p]<<8) ;
                if (( x < 320) && ( y < 240))
                    LCD_BUFFER[y][x] = c ;
                p=p+2;
            }
        }
}

void Lcd_Init(void)
```

```
{
    Lcd_Config() ;
    Lcd_Enable() ;
    Lcd_PowerEnable(1);
}
```

Timer模块主要用于产生1s定时。

timer.h文件内容如下：

```
#ifndef _TIMER0_H_
#define _TIMER0_H_

void Timer0_Init(void) ;


#endif
```

timer.c文件内容如下：

```
#include "timer.h"
#include "2440addr.h"
void Timer0_Init(void)
{
    rTCFG0 &= ~(0xff) ;
    rTCFG0 |= 99 ;
    rTCFG1 &=  ~(0xf) ;
    rTCFG1 |=  0X02 ;
    rTCNTB0 =  62500 ;       //1s 中断一次

    rTCON |= (1 << 1) ;      //手动更新
    rTCON = 0x09 ;           //自动加载，清除手动更新位，启动定时器

}
```

上述函数主要完成定时器0的初始化，在第8章“系统时钟和定时器”中曾进行过讲解。

interrupt.h文件内容如下：

```
#ifndef __INTERRUPT_H__
#define __INTERRUPT_H__

void Timer0_Interrupt_Init(void) ;

#endif
```

interrupt.c文件内容如下：

```
#include "2440addr.h"
```

```
void Timer0_Interrupt_Init(void)
{

      rINTMSK &=~(1 << 10) ;
}
```

这两个文件主要用于开启定时器0中断。读者也可以依此类推，其他类型的中断也可以使用这种方法处理。

isrservice.h文件内容如下：

```
#ifndef __ISRSERVICE_h__
#define __ISRSERVICE_h__

void Isr_Init(void) ;
void __irq Timer0_Isr(void) ;

#endif
```

isrservice.c文件内容如下：

```
#include"config.h"
#include"isrservice.h"

extern unsigned int flag ;

void Isr_Init(void)
{
      pISR_TIMER0 = (U32)Timer0_Isr ;

}

void __irq Timer0_Isr(void)
{
      Led1_On() ;
      flag ++ ;
      flag %= 5 ;
      rSRCPND |= 1 << 10 ;
      rINTPND |= 1 << 10 ;
}
```

这两个文件主要用于中断函数的安装，在第11章中曾经进行了详细的讲解。这里需要注意，在中断服务函数中只是对一个变量flag进行了简单的递增，每秒递增1次，该变量是在Main.c文件中定义的，所以在isrservice.c文件中使用了extern关键字。此外，flag%=5的意思将flag限定在1～5之间。

Main.c文件内容如下：

```
#include "timer.h"
#include"isrservice.h"
#include "uart.h"
#include "lcd.h"
#include "pic.h"
#include "pic1.h"
#include "pic2.h"
#include "pic3.h"
#include "pic4.h"
#include "led.h"
void IO_Init(void) ;

unsigned int flag = 0 ;

int Main(void)
{
    IO_Init() ;
    while(1)
    {
        switch(flag)
        {
            case 1:
                Paint_Bmp(pic) ;
                break ;
            case 2:
                Paint_Bmp(pic1) ;
                break ;
            case 3:
                Paint_Bmp(pic2) ;
                break ;
            case 4:
                Paint_Bmp(pic3) ;
                break ;
            case 5:
                Paint_Bmp(pic4) ;
                break ;
            default:

                break ;
        }
    }
```

```
        return 0;
}

void IO_Init(void)
{
        Timer0_Init() ;
        Timer0_Interrupt_Init() ;
        Isr_Init() ;
        Lcd_Init() ;
}
```

主函数中使用switch/case语句，根据flag的值进行显示不同的图片，因为flag的值每秒钟增加1，所以各幅图片轮流显示，到此为止可以理解isrservice.c文件中的flag%=5的含义：使flag的值限定在1～5之间，因为flag的每一个值对应一幅图片，总共有5幅图片。

15.3.4 实例测试

编译、链接生成.bin格式的二进制文件后下载到开发板，可以看到LCD上显示出了相应的图片，如图15-13、图15-14、图15-15所示。

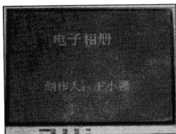


图 15-13 电子相册实验测试 1

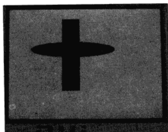


图 15-14 电子相册实验测试 2

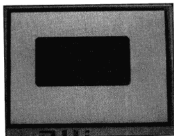


图 15-15 电子相册实验测试 3

15.3.5 实验总结

本实验主要向读者展示了电子相册的制作。在实际应用过程中，读者应根据具体的图片以及显示时间问题进行自行调整。

第 4 篇

理论知识扩展

第 16 章

嵌入式系统电源设计和 Linux 内核开发基础

电源是嵌入式系统的重要组成部分。电源设计的成败在很大程度上决定了系统设计的成败。电源设计需要从以下几个方面考虑：电压、电流、效率、噪声、纹波、体积、抗干扰等。对于采用电池供电的便携式嵌入式系统，还需要对系统的功耗进行管理。

很多初学者学习完 ARM 裸机开发后，需要进行 Linux 系统移植、Linux 核心编程等知识的学习。在本章中，笔者给出两个基本的 Linux 实验，帮助初学者理解 Linux 内核开发的基本流程。

16.1 直流稳压电源分类

按照调整管的工作状态来分，直流稳压电源可以分为以下两大类。

- 线性稳压电源

调整管工作在线性状态的直流稳压电源称为线性稳压器，线性稳压电源又分为两种：普通线性稳压器和低压差线性稳压器（Low DropOut Regulator，LDO）。

- 开关稳压电源

调整管工作在开关状态的直流稳压电源称为开关型稳压器，其基本原理是：首先储存能量，然后以受控方式释放能量，以获得所需的输出电压。开关式调整器升压泵采用电感器来储存能量，而电容式电荷泵采用电容器来储存能量。开关电源稳压器也可以分为两种：电容式 DC-DC 转换器（电荷泵）和电感式 DC-DC 转换器。

16.1.1 普通线性稳压器工作原理

线性稳压器是利用电压负反馈原理达到稳定输出电压的目的，即经误差放大器等组成的控制电路来控制调整管的管压降进而达到稳压的目的。普通线性稳压器的原理图如图 16-1 所示。

普通线性稳压器的特点是，输入电压必须大于输出电压，调整管工作在线性区。

具体工作原理：当输出电压 U_o 降低时，基准电压与取样电压的差值增加，比较放大器输出的电流增加，串联调整集电极和发射极之间的管压降减小，从而使输出电压升高；若

输出电压 U_o 超过所需要的设定值，比较放大器输出的电流减小，串联调整集电极和发射极之间的管压降增大，从而使输出电压降低。

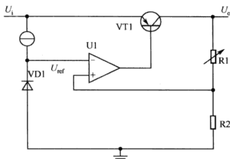


图 16-1 普通线性稳压器的原理图

普通线性稳压器有如下特点：

- 调整管功耗较大，电源效率低，一般只有45%左右。
- 体积大、需要占用较大的板子空间。
- 发热严重，要求较高的场合需要安装散热器。
- 静态电流较大，一般在 mA 级。
- 需要外接容量较大的低频滤波电容，增大了电源的体积。

普通线性稳压器价格低廉、静态电流大、效率较低、最小输入输出电压差较大，只能用于降压且对电源效率和体积没有严格要求的场合，如充电器、实验仪器等。

16.1.2 低压差线性稳压器工作原理

低压差线性稳压器的工作原理与普通线性稳压器的原理相同，都是采用电压负反馈来达到稳定输出电压的目的，即通过控制调整管上的压降变化来稳定输出电压。

低压差线性稳压器和普通线性稳压器的主要区别在于采用的调整管结构的不同，从而使 LDO 比普通线性稳压器压差更小，功耗更低。

低压差线性稳压器有如下特点：

- 稳定性好，负载响应快。
- 具有较小的输出电压纹波。
- 外围电路简单，一般只需要两个陶瓷滤波电容。
- 低静态电流，低功耗。
- 当输入/输出电压接近时可以达到很高的效率，当输入/输出压差较大时，效率会降低。

当系统输入、输出压差较小时，一般选用 LDO，因为当输入/输出压差较小时，LDO 可以达到较高的效率。例如，把锂电池电压（3.7～4.2V）转换为 3.3V 输出电压的应用中大多选用 LDO。

此外，LDO 具有较高的信噪比，因此常用做对噪声敏感的小信号处理电路供电；而且 LDO 没有开关时大的电流变化所引发的电磁干扰，很多手机、便携式设备等对干扰敏感的设备很多都采用 LDO 作为系统的电源芯片。

16.1.3 电容式开关电源的工作原理

电容式开关电源（电荷泵）的基本工作原理：利用电容的储能的特性，控制可控开关（双极型三极管或者 MOSFET 等）进行高频开关的动作，当开关闭合时，将输入的电能储存在电容里；当开关断开时，电能再释放给负载。

它的输出功率与占空比（由开关导通时间与整个开关的周期的比值）有关。电容式开关电源可以用于升压和降压的场合。

它内部的FET开关阵列以一定方式控制电容的充电和放电，从而使输入电压以一定因数（0.5、2或3）倍增或降低，从而得到所需要的输出电压。

电容式开关电源具有如下特点：

- 转换效率与输入电压密切相关。电荷泵的近似效率=U_{out}/U_{in}，因此当输出电压和倍率一定时，输入越小，电荷泵的效率越高。电荷泵效率一般可以达到75%以上。
- 输出电压一般是输入电压的倍数，常见的有±0.5倍压、±1倍压、±1.5倍压、±2倍压、±3倍压。
- 输出电流较小，一般在300mA以下。
- 具有较低的EMI和输出纹波。

对采用电池供电的便携式电子产品（蜂窝式电话、寻呼机、蓝牙系统和便携式电子设备）来说，采用电荷泵变换器来获得负电源或倍压电源，不仅减少了电池的数量、减小了产品的体积、重量，而且在减少能耗、延长电池寿命等方面起到极大的作用。

16.1.4 电感式开关电源的工作原理

利用电感的储能的特性，控制可控开关进行高频开关的动作。当开关闭合时，将输入的电能储存在电感里；当开关断开时，电能再释放给负载。它输出的功率或电压的能力与占空比（由开关导通时间与整个开关的周期的比值）有关。

电感式开关电源的特点：

- 功耗小，效率高。它通过使用低电阻开关和磁存储元件，极大地降低了转换过程中的功率损失，其效率可高达到96%。
- 稳压范围宽。从开关稳压电源的输出电压是由激励信号的占空比来调节的，输入信号电压的变化可以通过调频或调宽来进行补偿，这样，在工频电网电压变化较大时，它仍能够保证有较稳定的输出电压。所以开关电源的稳压范围很宽，稳压效果很好。
- 滤波的效率大为提高，使滤波电容的容量和体积大为减少。
- 电路形式灵活多样。有自激式和他激式，有调宽型（PWM）和调频型（PFM），有单端式和双端式等，设计者可以发挥各种类型电路的特长，设计出能满足不同应用场合的开关稳压电源。
- 可以输出大电流，静态电流小。
- 电感式开关电源存在较大的输出纹波和开关噪声。
- 需要的外围元件多，电路设计比较烦琐，特别是输出可调的开关电源，需要计算分压电阻、电感、滤波电容的取值。

电感式开关电源适用于输出电流较大、要求较高效率的电池供电的场合。

16.1.5 嵌入式系统设计中的电源芯片选型

在嵌入式系统设计中，电源芯片选型时可以从如下几个方面考虑。

- 确定输入、输出电压。根据输入、输出的大小关系选择降压、升压或升/降压芯片。如果是降压，则可以选择线性稳压器、电容式DC-DC（电荷泵）或降压DC-DC转

换器；如果是升压或者升/降压，则只能选择 DC-DC 转换器（电容式或者电感式升压 DC-DC 转换器）。如果是降压，考虑电源的效率，需要计算输入与输出之间的压差。若这个压差很小（远远小于 1V），则可以考虑选择低压差线性稳压器（LDO）；若这个压差在 1V 以上，则可以考虑选择普通线性稳压器或者电感式降压 DC-DC 转换器。如果对效率没有要求，在两种线性稳压器都可以的情况下，追求更低成本则可以选用普通线性稳压器。

- 在线性稳压器和 DC-DC 稳压器都可以的情况下，若把转换效率放在第一位，则可以选择 DC-DC 稳压器；若对价格限制得很严格，并且要求较小的纹波和噪声，则可以考虑选用线性稳压器。
- 在使用电池供电时，若要求较长的电池使用时间，需要优先考虑效率。无论是升压、降压、升/降压都可以选用 DC-DC 转换器。为获得较高的效率，此时需要参照 DC-DC 转换器芯片手册中的效率随负载电流变化曲线。TPS60210 芯片电源效率曲线如图 16-2 所示，可见，当输入电压为 1.8V，工作电流在 1～100mA 时，电源效率在 80%以上，但是当输入电压为 2.7V 时，电源效率在 60%左右。

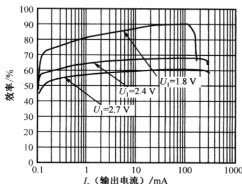


图 16-2 TPS60210 芯片电源效率曲线

- 为保证电池供电系统电源负荷变化较大应用的效率，最好选择 PFM/PWM 自动切换控制式 DC-DC 转换器。PWM 的特点是噪声低、满负载时效率高且能工作在连续导电模式下，PFM 具有静态功耗小，在低负荷时可改进稳压器的效率。当系统在重负荷时由 PWM 控制，在低负荷时自动切换到 PFM 控制，这样能够兼顾轻重负载的效率。在备有待机模式的系统中，采用 PFM/PWM 切换控制的 DC-DC 稳压器能够得到较高效率。这样的电源芯片有 TPS62110/62111/62112/62113 等。
- 留出一定的裕量。选用电源芯片时为保证电源的使用寿命，需要留有一定的裕量，较合适的工作电流为电源芯片最大输出电流的 70%～90%。如果用一个能输出大电流的稳压块来带动一个小电流的负载，虽然说驱动能力没有问题，但是可能会带来两个问题：一方面，成本会提高；另一方面，选用 DC-DC 转换器时效率可能会非常低，因为一般的 DC-DC 转换器在输出电流非常小或者非常大时效率都比较低。当使用线性稳压器（特别是普通线性稳压器）的时候，输出电流要尽量留出较多的裕量，因为线性稳压器的压降都消耗在稳压芯片上了，过大的负载电流会造成较为严重的发热，所以使用普通线性稳压器应该留有更大的裕量。
- 对于电池供电的系统，静态电流和效率是需要重点关注的参数。因为这直接关系到电池的使用寿命。静态电流是与负载电流大小几乎无关的消耗，越小越好。效率是能够转为有效利用能量多少的量度，同样容量大小的电池，电源的效率越高，静态电流越小，电池的寿命就越长。
- 输出电流大时应采用降压式 DC-DC 转换器。便携式电子产品大部分工作电流在

300mA 以下，并且大部分采用 5 号镍镉、镍氢电池，若采用 1～2 节电池，升压到 3.3V 或 5V 并要求输出 500mA 以上电流时，电池寿命不长或两次充电间隔时间太短，使用不便。这时采用降压式 DC-DC 转换器，其效率与升压式差不多，但电池充电间隔时间要长得多。

16.1.6 设计实例分析

在两节 5 号电池供电的 ZigBee 数据采集模块中，需要一个输出电压为 3.3V、电压波动在±5%以内、负载最大工作电流为 50mA 的电源，下面给出选型和设计的过程。

（1）确定输入输出电压范围。两节 5 号电池的标称电压为 3V，因此电源的输入电压范围为 0～3V，输出电压为 3.3V。所以需要选择能够进行升压的 DC-DC 转换器。

（2）选择转换器的类型。能够进行升压的芯片有电荷泵和电感式 DC-DC 转换器，电荷泵的输出电流比较小，电感式 DC-DC 转换器的输出电流比较大。但是，本系统中由于采集模块大部分时间是休眠，定时采集数据，所以，平均电流很小，无线发送数据时最大电流在 32mA 左右，所以选择电荷泵即可。

（3）查找符合条件的芯片。在业界比较大的电源厂商网站上寻找符合这样要求的芯片。通过查找发现，TI 和凌特有此类的芯片，如 TI 的 TPS60200、凌特的 LTC3430/3438/3440/3127 等。

（4）检查性能参数。仔细对各个芯片的效率、价格、输出电流、设计的复杂性等因素进行比较和分析，选出符合自己需求的电源芯片。经过比较和分析可知，选用 TPS60210 芯片可以满足系统要求。

（5）仔细阅读芯片手册，设计并绘制出符合要求的原理图。TPS60210 芯片数据手册给出的参考设计如图 16-3 所示。

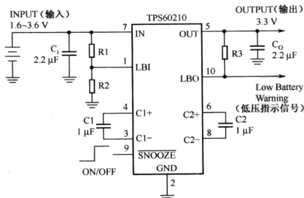


图 16-3 TPS60210 参考设计

其中，电阻 R1、R2 主要用于确定电池电压检测门限值。当电池电压低于该门限值时，10 号引脚 LBO 会给出低压指示信号。因此，最终绘制出合适的应用电路（如图 16-4 所示）。

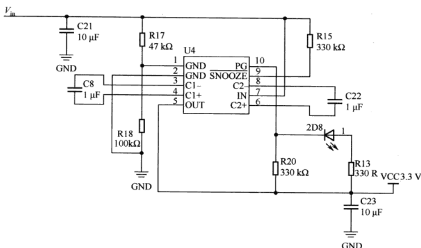


图 16-4　TPS60210 实际应用电路设计

16.2　Linux 内核基础实验

很多初学者学习完 ARM 裸机开发后，需要进行 Linux 系统移植、Linux 核心编程等知识的学习。下面给出两个基本的 Linux 实验，帮助初学者理解 Linux 内核开发的基本流程。学习以下实验需要读者对 Linux 内核有一定的了解。以下实验是在虚拟机平台上安装 Linux 操作系统完成的。

16.2.1　实验一：修改调度算法实验

Linux 2.6.14 内核采用的是 0（1）调度算法，该算法更倾向于交互型进程（如键盘输入数据时，响应速度较快）。下面修改调度算法，使其更具有公平性，即不再倾向于交互型算法。

当然，该实验没有什么实质性的用途，只是带领初学者了解 Linux 核心编程的基本流程。

1．实验目的

通过本实验，掌握 Linux 内核编译的方法和步骤，理解相应的测试方法。

2．实验内容

- 修改 Linux-2.6.14.4 内核进程调度算法。
- 编译 Linux-2.6.14.4 内核使其能正常启动。
- 编写相应的测试程序，运行测试程序，并对实验结果进行分析，得出实验结论。

3．实验原理

- 修改/kernel/sched.c 中的调度算法如下。

```
        if (!--p->time_slice) {
            ……
```

```
enqueue_task(p, rq->expired);
                if (p->static_prio < rq->best_expired_prio)
                        rq->best_expired_prio = p->static_prio;
        } else
                enqueue_task(p, rq->active);
    }
......
```

修改后的调度算法为：

```
if (!--p->time_slice) {
                ......
            enqueue_task(p, rq->expired);
    }
......
```

- 测试

编写相应的测试程序，对比调度算法更改前、后程序运行的情况。通过对比得出结论：修改后的调度算法不倾向于交互型进程。

4．实验步骤

（1）安装虚拟机 VMware 6.0.1，并在虚拟机上安装 Red Hat Enterprise 5（内核版本为 Linux-2.6.18）操作系统。

（2）下载 Linux-2.6.14.4 内核，并将其解压到/usr/src/Linux-2.6.14 目录下。

（3）编译 Linux-2.6.14.4 内核。

①修改 Linux-2.6.14/Kernel 目录下 sched.c 文件中 scheduler_tick()函数中的进程调度算法。

②修改后的调度算法为：

```
if (!--p->time_slice) {
......
enqueue_task(p, rq->expired);}
......
```

③进入 Linux-2.6.14 目录执行 make distclean。

④执行 make mrproper。

⑤执行 make menuconfig，因为是在虚拟机环境下编译，因此，应特别注意以下两项要正确配置。

- Device Drivers --->SCSI device support ---><*> SCSI disk support
- Device Drivers --->SCSI device support --->SCSI low-level drivers ---> <*> BusLogic SCSI support

⑥执行 make bzImage。

⑦执行 make modules。

⑧执行 make modules_install。

⑨执行 make install，该命令执行完后，进入/boot 目录可以看到 vmlinuze-2.6.14、

initrd-2.6.14.img 以及 System.map-2.6.14 已经生成；然后查看/etc/grub.conf 文件可看到启动信息，如图 16-5 所示。

```
title Linux-changed (2.6.14.4)
        root (hd0,0)
        kernel /vmlinuz-2.6.14.4 ro root=/dev/VolGroup00/LogVol00 rhgb quiet
        initrd /initrd-2.6.14.4.img
```

图 16-5　启动信息

系统重启后出现启动界面，如图 16-6 所示。选择“Linux-changed（2.6.14.4）”，系统能正常启动，说明移植成功。

```
GNU GRUB  version 0.97  (638K lower / 522176K upper memory)

Linux-changed (2.6.14.4)
Red Hat Enterprise Linux Server (2.6.18-8.el5)
```

图 16-6　启动界面

（4）编写测试程序。

①编写一个进程，大部分时间在执行循环，消耗 CPU 时间，用来模拟计算型进程，同时该进程还向一个文件 calculate.txt 中写入该进程的 PID 和该进程执行的起始时间和结束时间。

```
int main()
{
    int fd;
    char pidbuf[100] ="PID:";
    char pidtemp[5] ;
    time_t timer ;
    char *p ;
    int i, j  =  1000 ;
    itostr((int)getpid(),pidtemp) ;//store the pid to pidbuf
    strncat(pidbuf,pidtemp,4);

    timer  =  time(NULL) ;
    p  =  ctime(&timer) ;        //get the start time of the process
    strncat(pidbuf,p,strlen(p)) ;

    for(i = 0 ;i <3; i ++)
        while(j--) ;
    itostr((int)getpid(),pidtemp) ;//store the pid to pidbuf
    strncat(pidbuf,pidtemp,4);
    timer  =  time(NULL) ;
    p  =  ctime(&timer) ;
    strncat(pidbuf,p,strlen(p)) ;
    write(fd,p,strlen(p)) ; //get the end time of process
```

```
    fd  =  open("calculate.txt",O_CREAT | O_RDWR | O_APPEND,S_IRUSR | S_IWUSR) ;
    if(fd)
    {
        write(fd,pidbuf,strlen(pidbuf)) ;
    }
    close(fd) ;
    return 0 ;
}
```

②编写一个进程，向文件 iotestbuffer.txt 写入一串数据，用来模拟 I/O 型进程，同时该进程还向一个文件 iotime.txt 中写入该进程的 PID 和该进程执行的起始时间和结束时间。

```
int main()
{
    int fd,fd_test;
    char iobuffer[]="this is a   ioprocess test ,and used to modulate as IO test buffer , write ok!\n" ;
    char pidbuf[100] ="\nPID:";
    char pidtemp[5] ;
    time_t timer ;
    char *p ;
    int i ,j;
    itostr((int)getpid(),pidtemp) ;//store the pid to pidbuf
    strncat(pidbuf,pidtemp,4);
    timer  =  time(NULL) ;
    p  =  ctime(&timer) ;        //get the start time of the process
    strncat(pidbuf,p,strlen(p)) ;
    for(i=0 ; i<50 ; i++)
    for(j=0 ; j<100 ; j++)
    {
        fd_test=open("testbuffer.txt",O_CREAT | O_RDWR | O_APPEND,S_IRUSR | S_IWUSR) ;
        write(fd_test,iobuffer,strlen(iobuffer)) ;
        close(fd_test) ;
    }

    itostr((int)getpid(),pidtemp) ;//store the pid to pidbuf
    strncat(pidbuf,pidtemp,4);
    timer  =  time(NULL) ;
    p  =  ctime(&timer) ;
    strncat(pidbuf,p,strlen(p)) ;
    fd  =  open("iotest.txt",O_CREAT | O_RDWR | O_APPEND,S_IRUSR | S_IWUSR) ;
    if(fd)
    {
        write(fd,pidbuf,strlen(pidbuf)) ;
```

```
    }
    close(fd) ;
    return 0 ;
}
```

③编写一个进程，用来启动 50 个 I/O 型进程。

```
int main()
{
    int i = 5;
    pid_t pid ;
    while(i--)
    {
        if((pid=fork()) > 0)
        {
            continue ;
        }
        else if (pid == 0)
        {
            execl("./ioprocess",NULL) ;
        }
        else
        {
            printf("fork() error! please check!") ;
        }
    }
    exit(0);
}
```

④编写一个 shell 脚本，该脚本完成的功能：启动 50 个计算型进程，使其在后台运行，同时启动 50 个 I/O 型进程。

```
#!/bin/sh
for((i=0;i<50;i++));
do
    (./calculate)
done
    ./startio
```

5．实验结果测试

将测试程序分别在 linux-2.6.14、改变调度算法的 linux-2.6.14 内核上执行，其中 calculate.txt 和 iotime.txt 文件中记录了进程执行的时间信息。

测试 1，如表 16-1 所示。注：时间比 1 为 I/O 型进程执行时间与计算型进程的执行时间之比。时间比 2 为 I/O 进程执行时间占总执行时间的比值。

表 16-1　测试 1

Linux 内核版本		开始时间	结束时间	总时间	时间比 1(%)	时间比 2(%)
2.6.14	I/O 型进程	09:05:50	09:06:00	00:00:10	2.532	2.469
	计算型进程	08:59:15	09:05:50	00:06:35		
2.6.14（修改）	I/O 型进程	08:37:16	08:37:27	00:00:11	2.981	2.895
	计算型进程	08:31:07	08:37:16	00:06:09		

测试 2，如表 16-2 所示。注：时间比 1 为 I/O 型进程执行时间与计算型进程的执行时间之比。时间比 2 为 I/O 进程执行时间占总执行时间的比值。

表 16-2　测试 2

Linux 内核版本		开始时间	结束时间	总时间	时间比 1(%)	时间比 2(%)
2.6.14	I/O 型进程	09:13:42	09:13:54	00:00:12	3.061	2.970
	计算型进程	09:07:10	09:13:42	00:06:32		
2.6.14（修改）	I/O 型进程	08:46:09	08:46:18	00:00:09	2.639	2.571
	计算型进程	08:40:28	08:46:09	00:05:41		

测试 3，如表 16-3 所示。注：时间比 1 为 I/O 型进程执行时间与计算型进程的执行时间之比。时间比 2 为 I/O 进程执行时间占总执行时间的比值。

表 16-3　测试 3

Linux 内核版本		开始时间	结束时间	总时间	时间比 1(%)	时间比 2(%)
2.6.14	I/O 型进程	09:31:00	09:31:07	00:00:07	1.763	1.733
	计算型进程	09:24:22	09:30:59	00:06:37		
2.6.14（修改）	I/O 型进程	08:52:42	08:52:53	00:00:11	3.047	2.957
	计算型进程	08:46:41	08:52:42	00:06:01		

测试 4，如表 16-4 所示。注：时间比 1 为 I/O 型进程执行时间与计算型进程的执行时间之比。时间比 2 为 I/O 进程执行时间占总执行时间的比值。

表 16-4　测试 4

Linux 内核版本		开始时间	结束时间	总时间	时间比 1(%)	时间比 2(%)
2.6.14	I/O 型进程	09:41:16	09:41:26	00:00:10	2.525	2.463
	计算型进程	09:34:40	09:41:16	00:06:36		
2.6.14（修改）	I/O 型进程	09:24:55	09:25:04	00:00:09	2.601	2.535
	计算型进程	09:19:09	09:24:55	00:05:46		

测试结果分析：总体而言，修改调度算法后，计算型进程和 I/O 型进程的总执行时间减少了，这说明，修改调度算法后，实质上是提高了计算型进程的优先级，降低了 I/O 型

进程的优先级。因此，计算型进程和 I/O 型进程之间的切换减少了，CPU 用于进程调度的开销减少，总执行时间减少。

除第 2 个测试外，其他测试结果显示修改调度算法后，I/O 型进程执行时间与计算型进程执行时间的比值有所提高，I/O 型进程执行时间占总执行时间的比例也相应地提高了，即说明修改调度算法后，计算型进程可以尽快地执行。因此，系统总体上对 I/O 型进程的响应时间变长了。

由于测试是在虚拟机下安装 Linux 的，虚拟机和主机共享一个 CPU，对于测试 2 出现的测试结果，推测其原因可能是在测试期间，在主机上还进行文字编辑、上网等其他工作，因此，主机占用 CPU 的开销可能会大一些，造成虚拟机工作速度相对缓慢。

6. 实验结论

修改调度算法后，程序的总执行时间减少了，同时 I/O 进程执行时间相对延长了，这反映出修改调度算法后，进程调度不再倾向于 I/O 型进程，即不再倾向于交互型进程。

在实验过程中遇到的问题：将计算型进程推到后台执行需要在命令后面加&，但是其父进程还是当前终端 shell 的进程，而一旦父进程退出，则会发送 hangup 信号给所有子进程，子进程收到 hangup 以后也会退出。如果要在退出 shell 的时候继续运行进程，则需要将进程在一个 subshell 中执行，此时需要用括号将命令括起来，如$($command1 &)。

此外，在进行手动更改 grub.conf 文件时，出现开机无法启动的问题，但是使用 make install 后，grub.conf 文件会自动更改，同时所需文件会在/boot 目录下自动生成。

16.2.2 实验二：添加内核模块实验

1. 实验目的

通过本实验，掌握在 Linux 内核中添加动态模块的基本流程。

2. 实验原理

实验原理：在 hello 模块中定义一个全局变量，然后将其地址导出到内核符号表中，最后从另一个内核模块 print 中将该变量打印出来。

3. 实验步骤

（1）编写 hello.c 文件，内容如下：

```
#include<linux/kernel.h>
#include<linux/init.h>
#include<linux/module.h>
MODULE_LICENSE("GPL") ;
unsigned char *teststring = "the teststring defined in hello module!\n" ;
static int   moduleinit(void)
{
      printk(KERN_ALERT"hello-module load sucess!\n") ;
      return 0 ;
}
static void moduleexit(void)
```

```
{
    printk(KERN_ALERT"hello-module exit sucess!\n") ;
}

module_init(moduleinit) ;
module_exit(moduleexit) ;
```

EXPORT_SYMBOL(teststring) ;//将 teststring 导出到内核符号表中编写 hello 模块的 Makefile 文件，内容如下（2.6 内核 Makefile 文件的编写与以前版本略有不同）：

```
TARGET = hello
PWD := $(shell pwd)
KDIR := /lib/modules/$(shell uname -r)/build
obj-m := $(TARGET).o
default:
    make -C $(KDIR) M=$(PWD) modules
clean:
    $(RM) -f *.ko *.o *.mod.o *.mod.c *.symvers
```

（2）编写 print.c 文件，这时需要访问另一个模块中定义的变量，因此需要用 extern 关键字声明，内容如下：

```
#include<linux/kernel.h>
#include<linux/init.h>
#include<linux/module.h>
extern unsigned char *teststring ;//声明其他模块中定义的变量
MODULE_LICENSE("GPL") ;
int moduleinit(void)
{
    printk(KERN_ALERT"module-syscall load sucess!\n") ;
    printk(KERN_ALERT"%s\n",teststring) ;
    return 0 ;
}
void  moduleexit(void)
{
    printk(KERN_ALERT"module-syscall exit sucess!\n") ;
}
module_init(moduleinit) ;
module_exit(moduleexit) ;
```

（3）编写 print 模块的 Makefile 文件，内容如下：

```
TARGET = print
PWD := $(shell pwd)
KDIR := /lib/modules/$(shell uname -r)/build
obj-m := $(TARGET).o
```

```
default:
	make -C $(KDIR) M=$(PWD) modules
clean:
	$(RM) -f *.ko *.o *.mod.o *.mod.c *.symvers
```

分别编译 hello 模块和 print 模块，产生.ko 格式的文件，然后用 insmod 命令完成模块的安装，用 rmmod 命令实现模块的删除。需要注意的是，因为 print 模块需要访问在 hello 模块中定义的变量，因此，需要先安装 hello 模块，然后再安装 print 模块。

4．实验结果测试

通过上述步骤完成了 hello 模块和 print 模块的安装，通过 dmesg 命令可以查看到安装完 print 模块后输出了 teststring 指向的字符串的值，如图 16-7 所示。

```
[root@localhost hello]# insmod hello.ko
[root@localhost hello]# insmod ../moduletest/print.ko
[root@localhost hello]# rmmod ../moduletest/print.ko
[root@localhost hello]# dmesg

hello-module load sucess!
module-syscall load sucess!
the teststring defined in hello module!

module-syscall exit sucess!
```

图 16-7　测试结果

参 考 文 献

［1］王爽. 汇编语言［M］. 北京：清华大学出版社, 2008.
［2］谭浩强. C 语言程序设计［M］. 北京：清华大学出版社.2006.
［3］周立功, 张华. 深入浅出 ARM7［M］. 北京：北京航空航天大学出版社, 2005.
［4］王宇行. ARM 程序分析与设计［M］. 北京：北京航空航天大学出版社, 2008.
［5］韦东山. 嵌入式 Linux 应用开发完全手册［M］. 北京：人民邮电出版社, 2009.
［6］杜春雷. ARM 体系结构与编程［M］. 北京：清华大学出版社, 2008.
［7］Samsung.S3C2440Adatasheet［OL］.http://www.samsung.com/Products/Semico--nductor/FLASH/ TechnicalInfo/Datasheets.htm.
［8］孟海滨, 张红雨. 嵌入式系统电源芯片选型与应用［J］. 北京：单片机与嵌入式系统应用.2010 年 12 期.
［9］俞甲子, 石凡, 潘爱民. 程序员的自我修养［M］. 北京：电子工业出版社, 2009.
［10］Randal E.Bryant David R.O'Hallaron.深入理解计算机系统［M］. 北京：机械工业出版社, 2010.
［11］雷航, 王茜. 现代微处理器及总线技术［M］. 北京：国防工业出版社, 2011.
［12］Robert Love. Linux 内核设计与实现［M］. 北京：机械工业出版社, 2009.
［13］天嵌计算机科技有限公司. TQ2440 开发板使用手册, 2010.1.6.
［14］Findlay shearer. Power Management in Mobile Devices. 北京：机械工业出版社, 2010.